Java Real World Projects

A pragmatic guide for building modern Java applications

Davi Vieira

bpb

Java Real World Projects

A pragmatic guide for building modern Java applications

Davi Vieira

www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65898-972

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

Those who work late into the night and persist in their goals.

About the Author

Davi Vieira is a software craftsman with a vested interest in the challenges large enterprises face in software design, development, and architecture. He has over ten years of experience constructing and maintaining complex, long-lasting, and mission-critical systems using object-oriented languages. Davi values the good lessons and the software development tradition left by others who came before him. Inspired by such software tradition, he develops and evolves his ideas.

Davi started his career in technology by working as a Linux system administrator in the web hosting industry. After learning much about server task automation with shell scripting, he moved on to the banking industry, where he fixed bugs in legacy Java systems. Working with such legacy systems enabled Davi to face exciting challenges in a telecommunications organization, where he played a crucial role in helping the company adopt cloud-native development practices. Eager to learn and make a lasting impact, Davi works currently as a tech lead for an enterprise software company.

About the Reviewers

- **Ron Veen** is a seasoned software engineer with over 20 years of experience in the Java ecosystem. From mainframes to microservices, he's seen it all. His passion for software engineering and architecture drives his work.

  A certified Java expert (OCP and SCBCD/OCPBCD), Ron is proficient in a wide range of frameworks and libraries, from Apache to ZK. He is also keen on exploring alternative JVM languages like Kotlin.

  As a special agent and senior developer at Team Rockstars IT, Ron shares his knowledge as an international conference speaker and via his written works.

  He is the author of books on 'Java Cloud-native migrations with Jakarta EE' and 'virtual threads, structured concurrency, and scoped values'.

- **Stan Komar (Stanislaw Kazimierz Komar)** is a business owner focused on industrial automation and systems integration, using either Rockwell Automation, Schnieder, Siemens or Mitsubishi Tools. He has been actively programming since 1973 and is going strong. Although he should be retired, he enjoys the projects he works on so much that "Retirement can Wait." There is just so much change in the programming world that keeping up-to-date is very challenging. At 69 years young, he is an avid tennis player.

Acknowledgement

I vividly remember my brother Bruno Pinheiro de Oliveira exploring the computer, discovering things on the internet, and downloading Linux on a dial-up connection until late at night when I was a kid. My passion for technology started there. Thank you, my brother, for instilling such a passion in me.

I am grateful for my parents, Davi and Rosimar, who always believed in and supported my dreams, and also my wife, Eloise, and my son, Davi, for giving me a strong why when things got tough.

My Java journey started when learning the programming language was more about keeping my job during significant professional changes than anything else. I am still on this journey and want to acknowledge the people who contributed to my learning by allowing me to get along with them.

My first exposure to enterprise software at HSBC Global Technology was essential to lay the foundation that would culminate in the writing of this book. I am forever grateful for working with remarkable folks like Ana Carolina Moises de Souza, Renan Augusto da Silva, Caio Cesar Ferreira, Marco Aurélio Scheid, Ricardo André Pikussa, Carlos Bochnia, Emilio Fernandes, Adenir Rodrigues Filho, Alejandro Andrade, Jeferson Rodrigues (in memoriam), and Luiz Hauth from which will never forget his leadership lessons.

I cannot forget to mention Filipe Negrello for a brief, though intense, working relationship that led me to start working at Telefónica, where I had the chance to meet brilliant minds like Wilson Bissi, Lucas Maldaner, Mauricio Orozimbro de Souza, Raoni Gabriel, Roberto Silva Ramos Junior, Leonardo Henrique Pereira, Tiago Silvestrini, Renan Fabrão, Wagner Fernando Costa, Adriano Wierzbicki, Rodrigo Ribeiro, Nelmar Alvarenga, Hamilton Santos Junior, Jefferson Lira, Wagner Sales (in memoriam), Newton Dore, Giuliano Recco, Luiz Guilherme Mattos, Elizabete Yanase Hirabara Halas, André Santos, Arthur Gomes Junior, Renan Pazini, Thiago

Alberto Gil, and Julio Cesar Trincaus. Also, I want to thank Fagner da Silva and Paulo Jorge Lagranha for making a lasting impact on my career and Java journey that lasts to this day. I am immensely grateful for working with such amazing people who influenced me profoundly and shaped my character for my next challenge at SAP.

While working at SAP, I had the chance to collaborate on many cool projects that directly influenced the ideas I shared in this book. That is why I want thank Victor Fonseca, Vera Hillmann, Angelina Lange, Nils Faupel, Rafał Sokalski, Shumail Arshad, Saad Ali Jan, Kaiser Anwar Shad, Bruno Fracalossi Ferreira, Charne Elizabeth Pearson, Anna Kovtun, Rachel Falk, Arkadii Drovosekov, Reshmi Muhkerjeee, Anna Szarek, Rima Augustine, Babatunde Mustapha, Ahmed Alsharkawy, Taniya Vincent, and Torge Harbig. Thanks also to Vladimir Afanasenkov for being such a solid professional reference for me.

Last but not least, I want to thank Scott Wierschem, Bruno Souza, Elder Moraes, and Otavio Santana for every advice that influenced my thinking and motivated me to keep moving forward.

Preface

After so many years since its first release, Java remains as relevant as ever by powering the most critical applications in enterprises of all sizes. It is not uncommon to see Java applications still running ten, twenty, or more years ago, which serves as a testament to Java's robust and reliable nature. On the other side of the coin, Java continues to be the language of choice for many new projects that have the luxury of choosing from a set of high-quality frameworks like Spring Boot, Quarkus, or Jakarta EE, to name a few, that foster innovation and keep the Java language fresh to tackle the challenges of modern software development in the age of cloud and artificial intelligence. There has never been a better time to be part of such an exciting technological ecosystem, which offers endless opportunities to impact other people's lives through software.

Based on this landscape of opportunities and innovations, this book was written for those who decide to face the complexities, ambiguities, and hardships that may derive from any serious Java project. It is not meant to be a comprehensive guide for the Java programming language; instead, it takes a pragmatic approach in emphasizing, from the author's perspective, the relevant Java features and anything related to producing production-ready software.

Starting with exploring Java fundamentals, this book revisits core Java API components used to efficiently handle data structures, files, exceptions, logs, and other essential elements found in most enterprise Java applications. It also examines how modern Java features such as sealed classes, pattern matching, record patterns, and virtual threads can be used to create software systems that extract the best of what Java can provide. This book presents techniques for efficiently handling relational databases by tapping into the **Java Database Connectivity** (**JDBC**) and **Jakarta Persistence APIs** (**JPA**), thereby providing a solid Java data

handling foundation. Still, in the context of Java fundamentals, it explores how to increase overall code quality by employing unit and integration tests.

After covering the Java Fundamentals, this book explores how reliable software development frameworks such as Spring Boot, Quarkus, and Jakarta EE can be used to develop applications based on developer-friendly and productivity principles. These frameworks empower the Java developer by enabling him to use cutting-edge technology and industry standards that are the basis for most critical back-end applications.

With an eye on the everyday challenge of keeping Java applications running reliably in production, this book describes essential monitoring and observability techniques that help the Java developer better understand how well its application is behaving under different and quite often unexpected circumstances, putting the developer in a more pro-active than reactive position when dealing with bottlenecks, scalability and any other issue that can represent a risk for the application availability.

It concludes with an exploration of how different software architecture ideas, such as **domain-driven design** (**DDD**), layered architecture, and hexagonal architecture, can play crucial roles in developing change-tolerable and maintainable applications that not only deliver what the customer wants but also establish solid foundations that enable developers to gracefully introduce code changes with reduced refactoring efforts.

Chapter 1: Revisiting the Java API - This chapter revisits essential core Java APIs commonly seen in real-world projects. It starts by exploring the Collections API's data structures, showing the possible ways to handle data as objects in Java systems. Considering how often files must be dealt with, this chapter shows how to manipulate files using the NIO2. A closer examination of exceptions, followed by the Logging API, provides a solid foundation for helpful error handling and enhanced logging management. As most Java applications somehow need to deal with date and time, the Data-Time API is also explored. Closing the chapter, functional programming features such as streams and lambdas are covered to show how to write more efficient and concise Java code.

Chapter 2: Exploring Modern Java Features - Java is constantly

changing. Therefore, it is essential to keep up to date with its new features. This chapter looks into modern Java capabilities developers can leverage to build robust applications with sophisticated language features. It starts by explaining how to use sealed classes to increase inheritance control. It presents an intuitive way of matching Java types with pattern matching and using record patterns to extract data from matched types. Finally, it shows how to simplify the development of concurrent applications with virtual threads.

Chapter 3: Handling Relational Databases with Java - The ability to efficiently communicate with databases is a crucial characteristic of Java applications requiring persistence. Based on this premise, this chapter explores the **Java Database Connectivity** (**JDBC**) API, a fundamental Java component for handling relational databases. To enable developers to handle database entities as Java objects, it distills the main features of the Jakarta Persistence. The chapter finishes by examining local development approaches with container-based and in-memory databases for Java.

Chapter 4: Preventing Unexpected Behaviors with Tests - Automated tests help to prevent code changes from breaking existing system behaviors. To help developers with such an outcome, this chapter overviews two automated test approaches: unit and integration testing. It explores the reliable and widely used test framework JUnit 5. Finally, it describes how to use Testcontainers to implement reliable integration tests that rely on real systems as test dependencies.

Chapter 5: Building Production-Grade Systems with Spring Boot - Regarded as one of the most well-used Java frameworks, Spring Boot has withstood the test of time. So, this chapter explores the fundamental Spring components present in Spring Boot. Such fundamental knowledge leads to an analysis of how to bootstrap a new Spring Boot project and implement a CRUD application with major Spring Boot features.

Chapter 6: Improving Developer Experience with Quarkus - In the age of cloud, Quarkus has arisen as a cloud-first Java development framework with the promise of delivering a developer-friendly framework that empowers developers to create cloud-native applications based on the best industry standards and

technology. This chapter starts assessing the benefits Quarkus provides, shifting quickly to an explanation that details how to kickstart a new Quarkus project. It then shows how to use well-known Quarkus features to develop a CRUD application, including support for native compilation.

Chapter 7: Building Enterprise Applications with Jakarta EE and MicroProfile - Accumulating decades of changes and improvements, the Jakarta EE (formerly Java EE and J2EE) framework still plays a major role in enterprise software development. Relying on Jakarta EE, there is also MicroProfile, a lean framework to develop cloud-native microservices. Focusing on these two frameworks, this chapter starts with an overview of the Jakarta EE development model and its specifications. Then, it jumps to hands-on practice by showing how to start a new Jakarta EE project and develop an enterprise application. Finally, it shows how to create microservices using MicroProfile.

Chapter 8: Running Your Application in Cloud-Native Environments - Any serious Java developer must be able to make Java applications extract everything they can and perform well inside cloud environments. That is why this chapter explores cloud technologies, starting with container technologies, including Docker and Kubernetes. It explains how Java applications developed using frameworks like Spring Boot, Quarkus, and Jakarta EE can be properly dockerized to run in containers. Finally, it describes deploying such applications into a Kubernetes cluster.

Chapter 9: Learning Monitoring and Observability Fundamentals - Understanding how a Java application behaves in production while being accessed by many users is critical to ensuring the business health of any organization. Considering this concern, this chapter explores what monitoring and observability mean and why they are crucial for production-grade Java systems. It then shows how to implement distributed tracing with Spring Boot and OpenTelemetry. Also, it explains how to handle logs using Elasticsearch, Fluentd, and Kibana.

Chapter 10: Implementing Application Metrics with Micrometer - Metrics are essential to answer whether a Java application behaves as expected. Micrometer is a key technology that enables developers to implement metrics that answer those

questions. This chapter explains how to use a Micrometer to provide metrics in a Spring Boot application.

Chapter 11: Creating Useful Dashboards with Prometheus and Grafana - Visualizing important information regarding an application's behavior through dashboards can prevent or considerably speed up the resolution of incidents in Java applications. Based on such concern, this chapter examines how to capture application metrics using Prometheus. It then covers the integration between Prometheus and Grafana, two essential monitoring tools. Finally, it shows how to create helpful Grafana dashboards with metrics generated by a Java application and use Alertmanager to trigger alerts based on such metrics.

Chapter 12: Solving problems with Domain-driven Design - Based on the premise that the application code can serve as an accurate representation of a problem domain, **Domain-driven Design** (**DDD**) proposes a development approach that puts the problem domain as the driving factor that dictates the system's architecture, resulting in better maintainable software. Considering such maintainability benefits, this chapter starts with a DDD introduction, followed by an analysis of essential ideas like value objects, entities, and specifications. The chapter closes by exploring how to test the domain model produced by a DDD application.

Chapter 13: Fast Application Development with Layered Architecture - Enterprises of all sorts rely on back-end Java applications to support their business. The ability to fast deliver such applications is fundamental in a competitive environment. Layered architecture emerged organically among the developer community due to its straightforward approach to organizing application responsibilities into layers. In order to show the layered architecture benefits, this chapter starts with an analysis of the major ideas that comprise such architecture, followed by a closer look into applying layered architecture concepts in the development of a data layer for handling database access, a service layer for providing business rules, and an API layer for exposing system behaviors.

Chapter 14: Building Applications with Hexagonal Architecture - The pace at which technologies change in software systems has increased considerably over the last years, creating challenges for

those who want to tap into the latest cutting-edge innovations to build the best software possible. However, incorporating new technologies into existing working software may be challenging. That is where hexagonal architecture comes in as a solution to build change-tolerable applications that can receive significant technological changes without major refactoring efforts. With such advantage in mind, this chapter introduces the hexagonal architecture ideas, followed by hands-on guidance explaining the development of a hexagonal system based on the domain, application, and framework hexagons.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/bdae2b

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Java-Real-World-Projects**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.
Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Revisiting the Java API

Introduction

Structure

Objectives

Handling data structures with collections

Creating ordered object collections with lists

Providing non-duplicate collections with a set

Using maps to create key-value data structures

Using the NIO2 to manipulate files

Creating paths

Handling files and directories

Error handling with exceptions

Checked exceptions

Unchecked exceptions

Final block and try-with-resources

Creating custom exceptions

Improving application maintenance with the Logging API

Log handlers, levels, and formats

Exploring the Date-Time APIs

LocalDate

LocalTime

LocalDateTime

ZoneDateTime

Instant

Functional programming with streams and lambdas

Functional interfaces and lambda expressions

Predicates
Functions
Suppliers
Consumers
Streams
Sourcing streams
Intermediate operation
Terminal operation
Compiling and running the sample project
Conclusion

2. Exploring Modern Java Features
Introduction
Structure
Objectives
Getting more control over inheritance with sealed classes
Enforcing inheritance expectations
Increasing code readability with pattern matching
Introduction to pattern matching
Pattern matching for type
Pattern matching for switch statement
Pattern matching for record
Increasing application throughput with virtual threads
Understanding Java platform threads
Limitations of platform threads
Platform threads and blocking IO operations
Handling blocking IO with reactive programming
Writing simple concurrent code with virtual threads
Compiling and running the sample project
Conclusion

3. Handling Relational Databases with Java
Introduction

Structure
Objectives
Introduction to JDBC
Creating a database connection with the JDBC API
Getting a database connection with the DriverManager class
Getting a database connection with the DataSource interface
Executing simple queries with the Statement
Executing parameterized queries with the PreparedStatement
Calling store procedures with the CallableStatement
Processing results with the ResultSet
Simplifying data handling with the Jakarta Persistence
Defining entities
Defining entity relationships
OneToMany
ManyToOne
OneToOne
ManyToMany
Using Hibernate to handle database entities
Exploring JPQL and the Criteria API
Exploring local development approaches when using databases
Local development with a remote databases
Local development with in-memory databases
Local development with container databases
Compiling and running the sample project
Conclusion

4. Preventing Unexpected Behaviors with Tests
Introduction
Structure
Objectives
Overviewing unit and integration tests
Unit tests
Integration tests

Using JUnit 5 to write effective unit tests
Setting up JUnit 5
Introducing the account registration system
Testing the account registration system
The Arrange-Act-Assert pattern
Assertions
When to use Mockito
Setting up Mockito with JUnit 5
Adding an external call to the account registration system
Mocking external calls with Mockito
Executing tests with Maven
Implementing reliable integration tests with Testcontainers
Setting up Testcontainers
Integrating the account registration system with MySQL
Implementing an integration test with Testcontainers
Running integration tests with Maven
Compiling and running the sample project
Conclusion

5. Building Production-Grade Systems with Spring Boot
Introduction
Structure
Objectives
Learning Spring fundamentals
Using the Spring context to manage beans
Creating beans with the @Bean annotation
Creating beans with Spring stereotype annotations
The @Component annotation
The @Service annotation
The @Repository annotation
Using stereotype annotations
Injecting dependencies with @Autowired
Providing new application behaviors with aspects
Aspect

Jointpoint
Advice
Pointcut
Using the Spring AOP
Bootstrapping a new Spring Boot project
Creating a Spring Boot project with Spring Initializr
Implementing a CRUD application with Spring Boot
Setting up dependencies
Configuring the Spring Boot application
Defining a database entity
Creating a repository
Implementing a service
Exposing API endpoints with a controller
Sending HTTP requests to the Spring Boot application
Compiling and running the sample project
Conclusion

6. Improving Developer Experience with Quarkus

Introduction
Structure
Objectives
Assessing Quarkus benefits
Kickstarting a new Quarkus project
Building a CRUD app with Quakus
Injecting dependencies with Quarkus DI
Managed beans
Application-scoped beans
Singleton beans
Request-scoped beans
Persisting data with Hibernate
Setting up Quarkus to work with databases
Handling database entities with EntityManager
Simplifying database entity handling with Panache

Panache with repository pattern
Panache with active record pattern
Implementing an API with Quarkus REST
Writing native applications
Introducing the native image
Creating a native executable with Quarkus
Compiling and running the sample project
Conclusion

7. Building Enterprise Applications with Jakarta EE and MicroProfile

Introduction
Structure
Objectives
Overviewing Jarkarta EE
Designing multitiered applications
Client tier
Web tier
Business tier
Enterprise information system tier
Exploring Jakarta EE specifications
Jakarta EE Platform specification
Jakarta EE Web Profile specification
Jakarta EE Core Profile specification
Packing, deploying, and running Jakarta EE applications
Java Archive
Web Archive
Enterprise Archive
Introducing MicroProfile
Exploring MicroProfile specifications
Jakarta EE Core Profile specifications
MicroProfile specifications
Starting a new Jakarta EE project
Building an enterprise application with Jakarta EE

Adding microservices and cloud-native support with MicroProfile
Setting up the project
Defining a data source
Implementing a Jakarta Persistence entity
Implementing a repository with the EntityManager
Implementing a service class as a Jakarta CDI managed bean
Building API with Jakarta EE and MicroProfile
Using MicroProfile Health to implement health checks
Compiling and running the sample project
Conclusion

8. Running Your Application in Cloud-Native Environments
Introduction
Structure
Objectives
Understanding container technologies
Introducing virtualization
Full virtualization
Paravirtualization
Container-based virtualization
Exploring Docker
Learning Docker fundamentals
Managing Docker images
Creating Docker containers
Introducing Kubernetes
Kubernetes architecture
kube-scheduler
kube-apiserver
kube-controller-manager
Container runtime
kubelet
kube-proxy
Kubernetes objects

Pod
Deployment
Service
ConfigMap and Secret
Dockerizing a Spring Boot, Quarkus, and Jakarta EE application
Creating a bootable JAR of a Spring Boot application
Creating a bootable JAR of a Quarkus application
Creating a bootable JAR of a Jakarta EE application
Creating the Docker image
Deploying Docker-based applications on Kubernetes
Externalizing application configuration
Creating Kubernetes objects
Providing application configuration with a ConfigMap
Using a Secret to define database credentials
Deploying the application with a Deployment
Allowing access to the application with a Service
Using kubectl to install Kubernetes objects
Compiling and running the sample project
Conclusion

9. Learning Monitoring and Observability Fundamentals

Introduction
Structure
Objectives
Understanding monitoring and observability
Monitoring
Observability
Implementing distributed tracing with Spring Boot and OpenTelemetry
Building a simple distributed system
Configuring dependencies
Implementing the inventory service
Implementing the report service
Setting up Docker Compose, Jaeger, and Collector

Handling logs with Elasticsearch, Fluentd, and Kibana
Fluentd
Elasticsearch
Kibana
Setting up EFK stack with Docker Composer
Compiling and running the sample project
Conclusion

10. Implementing Application Metrics with Micrometer
Introduction
Structure
Objectives
Providing application metrics
Introducing Micrometer
Registry
Meters and tags
Counters
Gauges
Timers
Distribution summaries
Using Micrometer and Spring Boot to implement metrics
Setting up the Maven project
Configuring Spring Boot and Micrometer
Enabling metrics on the file storage system
Implementing the File entity
Implementing the File repository
Implementing the File metrics
Implementing the File service
Implementing the Controller class
Compiling and running the sample project
Conclusion

11. Creating Useful Dashboards with Prometheus and Grafana
Introduction

Structure
Objectives
Capturing application metrics with Prometheus
Learning the Prometheus architecture
Metrics exporters
Prometheus server
Metrics consumers
Getting Prometheus up and running
Downloading and installing Prometheus
Configuring Prometheus
Exploring the PromQL
Integrating Prometheus with Grafana
Configuring Prometheus as a Grafana data source
Creating Grafana dashboards with application-generated metrics
Building a Grafana dashboard
Visualization for the number of requests per HTTP method
Visualization for the file upload duration
Visualization for the download size
Triggering alerts with Alertmanager
Setting up the Alertmanager container
Defining Prometheus alerting rule
Defining Alertmanager notification channels
Compiling and running the sample project
Conclusion

12. Solving problems with Domain-driven Design
Introduction
Structure
Objectives
Introducing domain-driven design
Bounded contexts
Ubiquitous language

Event storming
Identifying event storm session participants
Preparing the event storm session
Domain events
Commands
Actors
Aggregates
The domain model
Conveying meaning with value objects
Expressing identity with entities
Defining business rules with specifications
Testing the domain model
Compiling and running the sample project
Conclusion

13. Fast Application Development with Layered Architecture
Introduction
Structure
Objectives
Importance of software architecture
Understanding layered architecture
A layer knows only the next layer
A layer can know other layers
Handling and persisting data in the data layer
Implementing the category entity and repository
Implementing the account entity and repository
Defining business rules in the service layer
Implementing the transaction service
Implementing the category service
Implementing the account service
Exposing application behaviors in the presentation layer
Implementing the transaction endpoint
Implementing the category endpoint

Implementing the account endpoint
Compiling and running the sample project
Conclusion

14. Building Applications with Hexagonal Architecture
Introduction
Structure
Objectives
Introducing hexagonal architecture
The domain hexagon
Entities
Value objects
Specifications
The application hexagon
Use cases
Input ports
Output ports
The framework hexagon
Input adapters
Output adapters
Arranging the domain model
Providing input and output ports
Exposing input and output adapters
Creating the output adapter
Creating input adapters
Compiling and running the sample project
Conclusion

Index

Chapter 1
Revisiting the Java API

Introduction

Java has been widely used in enterprises of all sorts of industries. One good example of how Java helps boost developer productivity is how it deals with memory management. While other programming languages may require developers to specify how program memory will be allocated, Java takes this memory allocation responsibility and lets the developers focus on the problem domain they want to solve. Another thing that can boost developer productivity even more is knowing how to use the Java API properly.

The Java API provides the language building blocks for application development. It crosses domains such as data structure handling with the Collections API, file manipulation with the NIO2 API, exception handling, and more. Instead of reinventing the wheel by implementing your data structures or algorithms, you can save a lot of time by tapping into what the Java API has to offer.

So, revisiting the Java API while observing how it can be used to solve problems commonly seen in software projects will give us a

solid foundation to explore further how Java can help us develop better software.

Structure

The chapter covers the following topics:

- Handling data structures with collections
- Using the NIO2 to manipulate files
- Error handling with exceptions
- Improving application maintenance with the Logging API
- Exploring the Date-Time APIs
- Functional programming with Streams and Lambdas
- Compiling and running the sample project

Objectives

By the end of this chapter, you will know how to use some of the most crucial Java APIs. With such knowledge, you will be able to solve common problems that may occur in your software project. Understanding the Java API is the foundation for developing robust Java applications.

Handling data structures with collections

Software development is a data processing activity in which we create systems that receive and produce data. As programmers, we are responsible for identifying the most efficient and straightforward ways to handle data in the applications we develop. Sometimes, this is not easy to achieve because what is efficient may not be simple, and vice versa. Developers have been searching for a balance between efficiency and simplicity, considering that nowadays, computing resources are not as expensive as they used to be.

The maintainability costs of a complex code that is optimized to be very efficient may be higher than a simpler code that consumes more computing resources but can be easily understood and changed.

It all depends on the context, of course. Certain businesses require their software to extract every bit of performance possible, like those of the trading and financial fields, where every millisecond, even nanosecond, can significantly influence monetary gains. Unless you are in such cases where high performance is required, there is always a trade-off space where building simpler, non-extremely efficient code may be the best option for your organization and customers.

As most applications do not fall under those requiring highly efficient performance, such applications can rely on standard data structures and algorithms for data processing. This is where the Java API comes in handy by providing a rich library called **Java Collections Framework** that offers a set of built-in data structures that suit most of the use cases we see in a typical back-end application.

Although the Java Collections Framework has interfaces and classes that may help those wanting to develop high-performance code, it really shines by providing support for frequently used data structures through the `java.util.List`, `java.util.Set`, and `java.util.Map` interfaces. The `Map` interface is not considered an authentic collection, but because we can handle map-based classes as if they were collections, we treat them as such.

Let us start our exploration with the `List` interface and some of its implementations.

Creating ordered object collections with lists

A common way to store objects in Java is through **lists**. The basic idea behind this data structure is that we can collect objects in an orderly manner, allowing us to preserve the sequence in which we store objects in a given list. It also enables duplicates, so if you try to insert the same object multiple times, it would not complain.

The `java.util.List` interface is derived from the `java.util.Collection` interface, meaning we can rely on the basic operations to handle a collection and the specific methods to manipulate lists. The `java.util.Collection` interface extends the `java.lang.Iterator` interface, which allows us to iterate through any list.

We have the **`java.util.ArrayList`** implementing the **`java.util.List`** interface. The **`ArrayList`** is certainly the most common list and data structure used in Java projects.

Based on the Java arrays, the **`ArrayList`** provides a list whose size can be dynamically increased, something we cannot do with ordinary Java arrays because their size, once defined, cannot be changed. Below is how we usually create an **`ArrayList`**:

```
List<String> listOfStrings = new ArrayList<>();
```

Since **`List`** and **`ArrayList`** are generic types, we can leverage type safety by specifying which object type our **`List`** will support. In the example above, we say that our **`List`** will accept only **`String`** objects. We establish it by declaring the variable type as **`List<String>`**. When creating the instance, we can explicitly define a type with angle brackets like **`ArrayList<String>`** or use the diamond operator **`<>`** as done in the example above with **`ArrayList<>`**. The compiler infers the **`ArrayList`** instance type by checking which type was defined for the variable **`String`** in our example.

Please note that we are using a **`List`** instead of an **`ArrayList`** as the **`listOfStrings`** variable type. It is a good programming practice to rely on interfaces. If we decide in the future that an **`ArrayList`** is no longer suitable for our needs, we can change the instance type without changing the variable type:

```
List<String> listOfStrings = new ArrayList<>();

listOfStrings = new LinkedList<>();
```

You may start your implementation using an **`ArrayList`** and have it referenced in multiple places of the codebase. Later, you may decide that a **`LinkedList`** is better for your use case. So, relying upon a **`List`** as the reference type may significantly decrease your refactoring efforts while changing the code to make use of a **`LinkedList`** instead of an **`ArrayList`**.

We can use a **`List`** to collect data from a file or database. It is a recurrent practice to get database records and represent them as objects in a **`List`**. If you have worked with Hibernate **Object–Relational Mapping** (**ORM**) or frameworks that depend on it, you may have seen helper methods allowing you to query databases and return results in a **`List`**.

When working with lists in general, we either get one already populated or we need to create and populate a new list. Regardless of the context, you may want to iterate over the list content:

```
List<String> listOfElements = List.of("Element 1",
"Element 2", "Element 3");

// Iterate using the index

for(var i = 0; i<listOfElements.size(); i++) {

  System.out.println(listOfElements.get(i));

}

// Iterate using the enhanced for-loop

for(var element : listOfElements) {

  System.out.println(element);

}

// Iterate using the forEach method with a lambda
expression

listOfElements.forEach(element ->
System.out.println(element));
```

We first iterate using the list indexes. Unless we have a special iteration logic that requires dealing with indexes, we will usually use the enhanced for-loop or the **`forEach`** method with a consumer lambda expression.

Note we are using **`List.of`** to generate our list; it is a helper method provided by the **`List`** interface that returns an immutable list. Once you create the list, there is no way to change its structure or its contents:

```
List<String> listOfElements = List.of("Element 1",
"Element 2", "Element 3");

listOfElements.set(0, "Element 4"); // it throws
java.lang.UnsupportedOperationException

listOfElements.add("Element 5");  // it throws
java.lang.UnsupportedOperationException

listOfElements.remove("Element 1");  // it throws
java.lang.UnsupportedOperationException
```

We cannot change the list contents nor add or remove an object.

If you intend to change the contents of your list, you can use the **`Arrays.asList`**:

```
List<String> listOfElements =
Arrays.asList("Element 1", "Element 2", "Element
3");

listOfElements.set(0, "Element 4"); // it works

listOfElements.add("Element 5");  // it throws
java.lang.UnsupportedOperationException

listOfElements.remove("Element 3"); // it throws
java.lang.UnsupportedOperationException
```

With **`Arrays.asList`**, it is only possible to change the list contents, not its structure. Any attempt to add or remove a list object will cause an **`UnsupportedOperationException`**.

The **`ArrayList`** is the way to go if you are looking for a completely mutable list. It allows adding and removing items:

```
List<String> listOfElements = new ArrayList<>();

listOfElements.add("Element 1");

listOfElements.add("Element 2");

listOfElements.remove("Element 1");

System.out.println(listOfElements.size()); // 1
```

In the example above, we remove a list item by providing an object of type **`String`** as a parameter, **`"Element 1"`**. It is also possible to remove a list item by specifying its index position:

```
List<String> listOfElements = new ArrayList<>();

listOfElements.add("Element 1");

listOfElements.remove(0);

System.out.println(listOfElements.size()); // 0
```

Removing using the index instead of object equality is always preferable regarding performance. When we remove objects from the list by passing an object for comparison, we must traverse all the list objects from beginning to end until we identify an object

that matches. This process takes up the computing time. When passing the index, it just removes the object in the specified index position.

Lists do not complain if you add duplicates:

```
List<String> listOfElements = new ArrayList<>();
listOfElements.add("Element 1");
listOfElements.add("Element 1");
System.out.println(listOfElements); // [Element 1, Element 1]
```

Adding duplicates to a list is okay because there is no mechanism to ensure uniqueness across the list contents. Now, if we want a data structure that excludes duplicates, then **`java.util.Set`** is the proper collection.

Providing non-duplicate collections with a set

The **`java.util.Set`** interface was conceived to provide a collection that excludes duplicates. However, we need to understand how an object can be compared to know what in Java can be considered a duplicate object. There are two ways an object can be compared in Java. The first way is equality based on the object's location in memory. Two objects are equal if they point to the same memory address location in the JVM. This is what happens when we compare objects using the **`==`** operator:

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
}
public class CheckEquality {
    public static void main(String... args) {
        Person person = new Person("john");
        Person samePerson = person;
```

```
        System.out.println(person == samePerson);
// true


System.out.println(person.equals(samePerson)); //
true

    }

}
```

We create a new **`Person`** instance and assign it to the **`person`** variable. We then create the **`samePerson`** variable pointing to the **`person`** variable. The result is **`true`** when comparing these variables using either the **`==`** operator or the **`equals`** method. The **`equals`** method comes from the **`Object`** class. All classes in Java inherit the **`equals`** method because every Java class extends the **`Object`** class. Below is how the **`equals`** method looks like in the **`Object`** class:

```
public boolean equals(Object obj) {

return (this == obj);

}
```

We can see that the **`equals`** method performs a comparison using the **`==`** operator. It means the default equality behavior of all objects we create in Java is based on the object location in memory.

Consider now the following scenario:

```
Person person = new Person("john");

Person samePerson = new Person("john");

System.out.println(person.equals(samePerson)); //
false
```

Comparing **`person`** and **`samePerson`** returns false because we have two objects with different memory locations. To solve this problem, we need a comparison mechanism that checks the object attributes instead of its memory location. We can accomplish this by overriding the **`equals`** and **`hashCode`** methods in the **`Person`** class:

```
@Override

public boolean equals(Object o) {
```

```
  if (this == o) return true;
  if (o == null || getClass() != o.getClass())
return false;
  Person person = (Person) o;
  return Objects.equals(name, person.name);
}
@Override
public int hashCode() {
  return Objects.hash(name);
}
```

Our **`equals`** implementation checks if both objects refer to the exact memory address location first. We do that using the **`==`** operator. Moving on, we check if the object being passed is **`null`** or if its class is different than the **`Person`** class. If the previous checks are okay, we cast the object to the **`Person`** type and compare the name attributes between the compared objects.

Note we are also overriding the **`hashCode`** method. This is especially important when dealing with **`Set`** collections because they are backed by a hash table that relies on the hash produced by the **`hashCode`** method to properly store objects in a key-value-based data structure where the hash number is the key and the object is the value. We will explore more about key-value data structures in the next session.

Our **`hashCode`** implementation produces a hash based on the name attribute of the **`Person`** class.

When we have **`equals`** and **`hashCode`** properly implemented, we can leverage the equality based on object attributes:

```
Person person = new Person("john");
Person samePerson = new Person("john");
System.out.println(person == samePerson); // false
System.out.println(person.equals(samePerson)); //
true
```

The equality check using the **`==`** operators returns `false` because

person and **samePerson** point to distinct objects located at different memory address locations. When we compare using **equals**, we get `true` because we now rely on our implementation that checks the **Person's** name attribute.

This knowledge about object equality in Java is fundamental to understanding how the **Set** interface excludes duplicates. All this discussion about object equality paves the way to solve any problem where we need to handle duplicate data.

One of the most used **Set** interface implementations is the **HashSet** class. Below is how we can create and add objects to a **HashSet**:

```
Set<Person> setOfPersons = new HashSet<>();

setOfPersons.add(new Person("john"));

setOfPersons.add(new Person("john"));

System.out.println(setOfPersons.size()); // 1
```

When we call add for the second time, the **HashSet** identifies that the **Person** object with the name attribute **"john"** already exists, ignoring its insertion. We can confirm the duplicate **Person** was not inserted by inspecting the **HashSet** size, which is one.

Sets, contrary to lists, do not guarantee collection ordering. So, the order in which you insert objects into a **Set** will not be preserved. You cannot remove objects from a **Set** collection using an index. A set is a key-value-based structure, so you must provide the object you want to remove. A hash code derived from a given object will be used as the key for lookup into the **Set**. If a key representing that object is found, then the object is removed from the **Set**:

```
Set<Person> setOfPersons = new HashSet<>();

Person person = new Person("john");

setOfPersons.add(person);

setOfPersons.remove(person);

System.out.println(setOfPersons.size()); // 0
```

Although a **Set** is backed under the hood by the key-value-based **Map** interface, we cannot use a **Set** to have a two-object relationship where one object is a key and the other is a value. So,

we need to rely directly on the **`Map`** interface and its implementations for that purpose. Let us see it in the next section.

Using maps to create key-value data structures

There are scenarios where we need more than just a list or a set of objects to solve our problems. In certain use cases, we may want a mapping structure that allows us to have an identification mechanism for our objects. For example, suppose you want to handle database row records and need to organize the data in a way that the record ID is connected to the row data it represents in the database. We can map the ID as the key and the row data as the value. Then, we can use the ID key to retrieve and manipulate row data values with such a mapping structure. That is, in a nutshell, what the **`Map`** interface provides.

Although the **`Map`** interface is not officially part of the Java Collections Framework, it has been treated as a collection because we can manipulate data produced by a **`Map`** in a way similar to how we manipulate data from other collections.

Imagine a scenario where we need to handle messages coming from a messaging system. Part of the handling involves receiving the messages and reconstructing them as objects, doing some processing with the message content, and then using the message ID to notify an external system that the message was handled.

1. Let us start by defining the **`Id`** class:

```
class Id {
    String id;
    Id(String id) {
        this.id = id;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
```

```
        Id id1 = (Id) o;
        return Objects.equals(id, id1.id);
    }
    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
    @Override
    public String toString() {
        return "Id{" + "id='" + id + '\'' + '}';
    }
}
```

Notice that we are overriding the **`equals`** and **`hashCode`** methods. That is important because duplicate keys are not allowed in a map, and we intend to use the **`Id`** as the key for our map.

2. Following the **`Id`**, we implement the **`Message`** class:

```
class Message {
    String content;
    Message(String content) {
        this.content = content;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
```

```
    }
}
```

We use the **`Message`** class to only store its content.

3. Finally, we create the **`HandleMessage`** class:

```
public class HandleMessage {
    public static void main(String... args) {
        Map<Id, Message> mapOfMessages = new HashMap
        mapOfMessages.put(new Id("MSG-1"), new Messa
        mapOfMessages.put(new Id("MSG-2"), new Messa
        mapOfMessages.put(new Id("MSG-3"), new Messa
        mapOfMessages.put(new Id("MSG-1"), new Messa

              mapOfMessages.remove(new Id("MSG-3"));


        System.out.println(mapOfMessages.size()); //
        mapOfMessages.forEach((id, message) -> {
            appendToContent(message);
            notifyProcessing(id);
        });
    }
    public static void appendToContent(Message messa
        var content = message.getContent();
        var processedContent = content + " - process
        message.setContent(processedContent);
    }
    public static void notifyProcessing(Id id) {
```

```
            System.out.println("Message "+ id + " proces
        }
    }
```

Executing the above code will produce the following output:

```
Map size: 2
Message Id{id='MSG-2'} processed.
Message Id{id='MSG-1'} processed.
```

In the **HandleMessage** class, we create an empty **HashMap**, the most common **Map** interface implementation. The **Map<Id, Message>** type specifies that we have the **Id** class as the key and the **Message** class as the value. Next, we add four map entries and remove one entry. Notice we add two entries with the same id, **"MSG-1"**. When we try to add the **"MSG-1"** entry for the second time, the insertion is ignored because the map already contains an entry with such a key. Then, we remove the entry **"MSG-3"** from the map. Ignoring duplicates and removing an entry using the key is possible because of our **equals** and **hashCode** implemented earlier in the **Id** class. Finally, we print the map size, which is two.

Proceeding with the code, we use **forEach** to iterate over the map entries. The syntax we use here is a lambda expression, which we will explore later in this chapter. The map processing is straightforward; we call **appendToContent(Message message)** and append the **" - processed"** to the original message, followed by a call to **notifyProcessing(Id id)**, where we use the **Id** to notify the message was processed.

The previous example illustrates a typical pattern where we map data coming from somewhere, perform some data processing, and finally take further action after finishing with data processing. Instead of using a **List** or **Set**, we chose a **Map** to map the **Id** to a **Message**, allowing us to conveniently use each object for different purposes.

The Java Framework Collections is an extensive topic with much more than we covered in this session. Going into every aspect of this topic is out of this book's scope. So, this session focused on collection approaches commonly seen in Java projects. Let us see

now how we can efficiently handle files using Java.

Using the NIO2 to manipulate files

Numerous cases can be where a Java application needs to interact with the operating file system to handle files and directories. The author recalls a project where they worked with a system that generated reports and made them available through files for download. The application would run logic to create the report and save it somewhere in the file system where the JVM was running. After finishing, the application would provide a URL so the user could download the report file.

Depending on your use case, you may often handle files and directories with Java. The good news is that Java has a powerful I/O API called NIO.2 (Non-blocking I/O, version 2), allowing an intuitive and smooth file system management experience. We can find the NIO.2 API in the `java.nio` package. There is also the standard I/O API found in the `java.io` package, but this is the old way to handle files in Java. New applications should rely on the NIO.2 API.

Our focus in this section will be on the NIO.2 API because it provides modern features to handle files and directories.

Creating paths

A file system represents how files and directories are organized in an operating system. Such representation follows a tree-based structure where we have a root directory and other directories and files below it. We have two kinds of file systems, one deriving from Windows and another from Unix (e.g., Linux or macOS). On Windows, the root directory is usually `C:`, while on Linux, it is `/`.

How we separate paths can also change depending on the operating system. For example, on Windows, we use the backslash `\`, while on Linux, it is the forward slash `/`.

The following is how we can identify the possible root directories available for a Java application:

```
System.out.println(FileSystems.getDefault()); //
sun.nio.fs.LinuxFileSystem
```

```
FileSystems.getDefault().getRootDirectories().forEach(Syste
// "/"
```

Since the above code is executed in a Linux machine, the `FileSystems.getDefault()` returns `sun.nio.fs.LinuxFileSystem`. When checking for the possible root directories, it gets only the `/`, which is the root directory in Linux systems.

The NIO.2 API provides the `Path` interface representing paths in a file system. A path can be either a file or a directory. There is also the concept of symbolic links, which resolve to a file or directory, so Java allows us to create `Path` objects using symbolic links.

The following is how we can create a `Path` object:

```
Path path = Path.of("/path/example");
```

We use the factory method `Path.of` with a `String` representing the path we want to create. One important thing to understand here is that the path `String` you pass to the `Path.of` may not exist. So, creating a new `Path` object does not mean the path exists in the operating system.

Paths can be absolute or relative. We call absolute every complete path, meaning it contains all path components, including the root directory. The following are examples of absolute paths:

```
Path.of("/home/john/textFile.txt");  // Linux
absolute path

Path.of("C:\\users\\john\\textFile.txt"); // Windows
absolute path
```

The absolute path above comprises a file called `textFile.txt` and three directories, including the root directory. Note we are using a double backslash for the Windows path; this is required because one backslash is interpreted as the escape character.

Relative paths are those partially representing a path location:

```
Path.of("./john/textFile.txt"); // Linux relative
path

Path.of(".\\john\\textFile.txt"); // Windows relative
path
```

We use `./` or `.\` to indicate the current directory. In the example

above, we are not providing the complete path where the **`textFile.txt`** is located, making the path a relative one.

The NIO.2 API offers helpful factory methods to manipulate the **`Path`** objects in different ways. We can, for example, combine an absolute path with a relative one:

```
Path absPath = Path.of("/home");

Path relativePath = Path.of("./john/
textFile.txt");

Path combinedPath = absPath.resolve(relativePath);

System.out.println(combinedPath); // /home/./john/
textFile.txt

System.out.println(combinedPath.normalize()); // /
home/john/textFile.txt
```

The **`resolve`** method called in a **`Path`** object allows us to combine it with another **`Path`** object as we did with the **`absPath`** and the relative path.

When printing the **`combinePath`** for the first time, we see the presence of the **`./`** (current directory) path element, which is redundant and can be excluded from the path representation. When printing the result of calling the **`normalize`** method for the **`combinedPath`**, we see the normalized **`combinedPath`** without the **`./`** element. The **`normalize`** method is helpful to clean up paths containing redundant elements such as **`./`** (current directory) and **`../`** (previous directory).

Now that we know how to manipulate **`Path`** objects let us see how to use them to handle files and directories.

Handling files and directories

The NIO.2 API provides the **`Files`** class containing factory methods to handle files and directories. The following is an example of how we can use this class:

```
import java.io.IOException;

import java.nio.file.*;

public class HandlingFilesAndDirectories {
```

```
    public static void main(String... args) throws
IOException {
        Path textFile =
Files.createFile(Path.of("/tmp/textFile.txt"));
        String content = """
             First line
             Second line
             Third line
             """;
        Files.writeString(
                textFile,
                content);
    }
}
```

We start by creating a new file with **`Files.createFile(Path.of("/tmp/textFile.txt"))`**. Observe that the **`createFile`** method requires a **`Path`** object with the string representing our file location. A **`FileAlreadyExistsException`** is thrown if the file already exists. Then, we create a string text block with three lines. Finally, we write these lines into our file using **`Files.writeString`**. The **`writeString(Path path, CharSequence csq)`** method expects the following parameters:

- A **`Path`** object representing the file we want to write.
- A **`CharSequence`** (**`String`** implements it) representing the data we want to write.

The following is the output of the **`/tmp/textFile.txt`** file:

```
First line
Second line
Third line
```

With the following code, we add the fourth line to the text file:

```
String content = "Fourth line";
```

```
Files.writeString(
        Path.of("/tmp/textFile.txt"),
        content,
        StandardOpenOption.APPEND);
```

The **`StandardOpenOption.APPEND`** is an option that allows writing data to a file without overwriting already existing content. It appends new lines starting from the last file line.

You may want to create directories and move files across them. The following example shows how we can do that using the NIO.2 API:

```
Path dirA = Path.of("/tmp/dirA");
Path fileA = Path.of("/tmp/fileA");
System.out.println(Files.isDirectory(dirA)); //
false
System.out.println(Files.isRegularFile(fileA)); //
false


Files.createDirectory(dirA);
Files.createFile(fileA);
System.out.println(Files.isDirectory(dirA)); //
true
System.out.println(Files.isRegularFile(fileA)); //
true


Path destFilePath = Path.of(
        dirA +
        FileSystems.getDefault().getSeparator() +
        fileA.getFileName()
);
System.out.println(Files.move(fileA,
destFilePath)); // /tmp/dirA/fileA
```

We start by creating the **`Path`** objects that **`dirA`** and **`fileA`**

represent in our directory and file, respectively. We can confirm that the directory and file do not exist by using the **`isDirectory`** and **`isRegularFile`** methods that check in the file system if the path provided exists. Following, we call the **`File.createDirectory`** and **`File.createFile`** methods with the **`Path`** objects **`dirA`** and **`fileA`** created previously. Executing such methods will create the directory and file into the file system. We can confirm it by calling the **`isDirectory`** and **`isRegularFile`** methods with the **`dirA`** and **`fileA`** Path objects. Finally, we create a **`Path`** object **`destFilePath`** that represents the target place where we want to move our file, created on **`"/tmp/file"`**. We construct the string for **`destFilePath`** using:

- The target directory that we get from **`dirA`**.
- The platform-dependent path separator we get by galling **`getSeparator`**, which for Linux systems is **`/`** and Windows usually is C.
- The filename element we get from the **`fileA`** path.

Calling **`Files.move`** moves the file to the desired destination and returns the target path where the file is now located, which is **`/tmp/dir/file`**. There is also **`Files.copy`**, which works similarly to **`Files.move`**.

The NIO.2 API provides helpful factory methods to read file contents like, for example, the **`Files.lines`** method:

```
Path newFile = Path.of("/tmp/newFile.txt");
String content = """
First Line
Second Line
""";
Files.writeString(
        newFile,
        content,
        StandardOpenOption.CREATE);
Files.lines(newFile).forEach(System.out::println);
```

The code creates a text file with two lines. Note that we are now using the option `StandardOpenOption.CREATE` that creates a new file if it does not already exist. Then we call `Files.lines(newFile)` that returns a `Stream<String>` that we can use to read line by line from our file. Using a stream to read text files can be efficient in terms of memory usage because streams do not keep the whole file content in memory. The stream allows us to read line by line without compromising memory resources.

The next section will explore how to use exceptions to deal with unexpected behavior in a Java application.

Error handling with exceptions

Developing a robust Java application means understanding what can go wrong and how the application can gracefully handle unexpected behaviors. Your application can have a logic that breaks whenever a user provides an unusual input that the application is unprepared to handle. When integrating your application with a database, you assumed the data would always come in one format until it came in a different format. The API you are consuming may suddenly change its contract, causing trouble for your application. The file system where your application saves files may become full, causing failure when saving new files. We can go on and on with the things that can cause issues in an application.

The Java architects were aware of the failure conditions an application may face, and because of that, they designed an error-handling mechanism based on the `Throwable` class. This class handles all errors and exceptions while a Java application is running.

In Java, we have checked and unchecked exceptions. Checked exceptions represent failure conditions that a Java application must handle or catch, while unchecked exceptions are the opposite, meaning they are not supposed to be caught even though the programmer may catch them if he wants. We can create a checked exception by using the `Exception` class directly or extending it. There is also a `RuntimeException` class, which is considered an unchecked exception, extending the `Exception` class. Every other class that extends the `RuntimeException` can also be considered an unchecked exception. The `Error` class extends directly from the

Throwable class and is considered an unchecked exception.

Catching a checked exception allows us to provide some graceful recovery mechanism to our application while also notifying us about what went wrong. Properly using the exception handling mechanism in Java can significantly improve error troubleshooting efforts by allowing us to identify faster why things are not working as expected in our application.

Next, let us check some use cases for checked exceptions.

Checked exceptions

Here, we have an example showing how checked exceptions can be used:

```
public class ExceptionAnalysis {
    public static void main(String[] args) {
        try {
            checkParameter(-1);
        } catch (Exception e) {
            System.out.println("The following
error occurred: "+
            e.getMessage());
            e.printStackTrace();
        }
    }

    public static void checkParameter(int
parameter) throws Exception {
        if(parameter<0)
            throw new Exception("Negative numbers
are not allowed");
    }
}
```

In Java, we use the **try-catch** construct to handle exceptions.

Inside the **`try`** block, we put the code that may **`throw`** a checked exception. We may handle the exception inside the **`catch`** block by printing a friendly error message or doing something to recover from a failure condition.

The code above will print the following output:

```
The following error occurred: Negative numbers not allowed

java.lang.Exception: Negative numbers not allowed

 at ExceptionAnalysis.checkParameter(ExceptionAnalysis.java:14)

 at ExceptionAnalysis.main(ExceptionAnalysis.java:5)
```

The **`checkParameter`** checks if the provided parameter is a negative integer. If true, it throws an exception using the **`throw`** keyword. The **`Exception`** class provides a one-parameter constructor, allowing us to provide a **`String`** message to the exception.

Since we are throwing a checked exception, we must either handle it inside the **`checkParameter`** or put a **`throws`** keyword on the method declaration. The **`throws`** keyword specifies one or more exceptions the method may throw.

We use the **`throws`** keyword, which means any other method calling our method will have to either handle the exception or include the throws keyword in its declaration. If the programmer does not explicitly handle the exception, then the default JVM exception handler handles it.

Unchecked exceptions

Contrary to checked exceptions, unchecked exceptions are not required to be handled at runtime by our code. Although not recommended, handling unchecked exceptions is legal.

Following is an example showing how an application can throw an unchecked exception:

```
var listOfStrings = List.of("a","b");

var first = listOfStrings.get(0);

var second = listOfStrings.get(1);
```

```
var third = listOfStrings.get(2);  //
IndexOutOfBoundsException: Index: 2 Size: 2
```

The **`IndexOutOfBoundsException`** is considered an unchecked exception because it extends the **`RuntimeException`** class. It occurs when we try to access an index outside the list bounds. The default JVM exception handler handles this exception directly.

When handling exceptions, there are scenarios where we want to execute some code whenever the application leaves a **`try-catch`**. That is when the **`final`** block comes into play, allowing us to always execute some code, regardless of whether an exception is caught.

Final block and try-with-resources

Aside from the **`try-catch`**, there are also the **`try-finally`** and **`try-catch-finally`** constructs. The **`finally`** block represents code always executed due to handling an exception. **`finally`** blocks are frequently used when we want to ensure the application closes resources used by system calls executed inside the **`try`** block. The following example illustrates such a scenario:

```
public void writeToFile() throws IOException {
FileOutputStream fileOutputStream = new
      FileOutputStream("file.data");
DataOutputStream dataOutputStream = new
      DataOutputStream(fileOutputStream);
try {
            dataOutputStream.writeChars("Some text
data");
      } catch (IOException e) {
            e.printStackTrace();
      } finally {
            dataOutputStream.close();
            fileOutputStream.close();
      }
```

```
}
```

The code above serializes text characters into bytes using the **`DataOutputStream`**, which writes the data using the file provided by the **`FileOutputStream`**. We are catching **`IOException`** because this is a checked exception thrown by the **`writeChars`** method from the **`DataOutputStream`**. Then, we have the **`finally`** block, where we close the resources we opened with **`FileOutputStream`** and **`DataOutputStream`** outside the **`try-catch-finally`** block at the beginning of the method.

Closing resources is always recommended, especially when opening database connections or reading or writing data with input and output streams.

Java provides another approach that makes our code more concise; shown as follows:

```
public static void writeToFile() {
try (FileOutputStream fileOutputStream = new
            FileOutputStream("file.data");
     DataOutputStream dataOutputStream = new
            DataOutputStream(fileOutputStream))
{
    dataOutputStream.writeChars("Some text");
} catch (IOException e) {
          e.printStackTrace();
}
}
```

The code above uses the **`try-with-resources`** construct, which declares resources between parentheses in the try block initialization. All the resources are closed automatically in a **`try-with-resources`** block, but this only happens if the class opening the resource implements either the **`Closable`** or **`AutoClosable`** interfaces.

Before finishing this exception discussion, let us examine how we can create exceptions.

Creating custom exceptions

The Java language comes with helpful checked and non-checked exceptions we can use in our applications. Although we could use the **`Exception`** class everywhere in our system to tell if something went wrong, that is not a good practice and can create ambiguities because of different error conditions using the same **`Exception`** class.

To provide more efficient and accurate error handling, we can extend the **`Exception`** class to create checked exceptions or the **`RuntimeException`** class to create unchecked exceptions.

The following is how we create a customized checked exception:

```
public class NegativeNumberException extends
Exception {

    public NegativeNumberException(String message)
{

        super(message);

    }

}
```

We can create checked exceptions by extending the **`Exception`** class and providing no additional constructors. The **`NegativeNumberException`** class goes beyond that and provides a constructor receiving a **`String`** message parameter, allowing us to provide a meaningful message when throwing the exception:

```
throw new NegativeNumberException("Negative numbers
are not allowed");
```

Instead of throwing a generic **`Exception`**, we can throw the **`NegativeNumberException`** to express more accurately a failure condition where the application cannot handle negative numbers, for example.

Creating, throwing, and handling custom exceptions contributes to developing more straightforward troubleshooting applications. Another thing that can also support troubleshooting is how we log application behaviors. Let us investigate it in the next section.

Improving application maintenance with the

Logging API

When developing new applications, our main concern may be first solving the problem. It usually means quick and dirty code. It is not the final solution, but it does the job. At this stage, we find ways to improve the code by cleaning it and improving the performance where possible. Another thing that we consider is how we will log the different application behaviors required to enable the solution. After all, providing relevant log messages can significantly help when investigating system issues.

To help us provide better logging capabilities to our applications, Java has the Logging API, which allows us to log application behaviors with different log handlers, levels, and formats. In the following section, we explore how they work.

Log handlers, levels, and formats

The Logging API is based on log handlers, levels, and formats. We can use built-in handlers like the **`ConsoleHandler`** and **`FileHandler`**, but we can also create customized log handlers. These handlers are responsible for sending the log messages somewhere, like a log file, for example, in the case of a **`FileHandler`**.

We can use the **`logging.properties`** configuration file found in the **`JAVA_HOME/conf`** directory for Java versions after 8 or in the **`JAVA_HOME/jre/lib`** directory for Java versions before 8. It is also possible to define a different location where the configuration file will be located using the following system property when executing the Java application:

```
java -Djava.util.logging.config.file=/tmp/
logging.properties
```

The following is how we can configure the **`logging.properties`** file:

```
handlers = java.util.logging.FileHandler,
java.util.logging.ConsoleHandler

java.util.logging.FileHandler.pattern=%t/java-
severe-%u.log

java.util.logging.FileHandler.level=WARNING
```

```
java.util.logging.FileHandler.formatter=java.util.loggin
```

Only the **ConsoleHandler** is enabled by default. The configuration above also includes the **FileHandler** through the handler's property. The pattern property specifies how Java will create the files containing logging data. We use **%t** to store log files in the operating system's temporary directory, which in Windows is **C:\TEMP**, and in Linux is the **/tmp** directory. We then set the level property as **WARNING** for the **FileHandler**. Here are all the available levels we can set:

OFF

SEVERE

WARNING

INFO

CONFIG

FINE

FINER

FINEST

ALL

OFF means no level at all, and **ALL** means all log levels. The level mechanism works so that the **FINEST** log is the lowest possible configuration, whereas the **SEVERE** is the highest. Suppose we configure a handler with the **FINEST** level. In that case, it will capture **FINEST** log level messages and other messages with a level above it, like **FINER**, **FINE**, and so on, until the **SEVERE** level. Conversely, if we configure a handler with the **WARNING** level, it will capture only messages with **WARNING** and **SEVERE** levels, ignoring all messages defined with levels below those two.

Finally, we have the formatter property, which defines how we display log messages. In the example above, we use a built-in format called **java.util.logging.SimpleFormatter**, but we can also implement our custom format by extending the **java.util.logging.Formatter** class.

Following, we have a program that logs messages using the **logging.properties** file that we defined previously:

```
package org.corp;
```

```
import static java.util.logging.Level.INFO;
import static java.util.logging.Level.SEVERE;
import java.io.IOException;
import java.util.logging.Logger;
public class TestApp {
    private static final Logger logger =

Logger.getLogger(org.corp.TestApp.class.getName());
    public static void main(String... args) throws
IOException {
        logger.log(INFO, "Start operation");
        try {
            execute();
        } catch (Exception e) {
            logger.log(SEVERE, "Error while
executing operation", e);
        }
        logger.log(INFO, "Finish operation");
    }
    public static void execute() throws Exception
{
        logger.log(INFO, "Executing operation");
        throw new Exception("Application
failure");
    }
}
```

We obtain a **`Logger`** object by executing the following line:

```
Logger logger =
Logger.getLogger(dev.davivieira.Main.class.getName());
```

We use this object to log messages. It is also possible to set the

default log level for the entire class by executing something like the below:

```
logger.setLevel(Level.INFO);
```

Setting such a configuration means the logger will capture all messages logged with **SEVERE**, **WARNING**, and **INFO** levels. Other logging levels below **INFO**, like **CONFIG** and **FINE**, would be ignored.

Executing the **TestApp** program will cause the creation of the file **/tmp/java-severe-0.log** with the following content:

```
Jan 28, 2024 12:24:44 PM org.corp.TestApp main
SEVERE: Error while executing operation
java.lang.Exception: Application failure
 at org.corp.TestApp.execute(TestApp.java:26)
 at org.corp.TestApp.main(TestApp.java:17)
```

The **FileHandler** provides the above output. Notice we only see **SEVERE** log messages. The **INFO** log messages are omitted because the **FileHandler** log level is set to **WARNING**, which means we display only **WARNING** and **SEVERE** log messages.

The folllowing is the output provided by the **ConsoleHandler**:

```
Jan 28, 2024 12:24:44 PM org.corp.TestApp main
INFO: Start operation
Jan 28, 2024 12:24:44 PM org.corp.TestApp execute
INFO: Executing operation
Jan 28, 2024 12:24:44 PM org.corp.TestApp main
SEVERE: Error while executing operation
java.lang.Exception: Application failure
 at org.corp.TestApp.execute(TestApp.java:26)
 at org.corp.TestApp.main(TestApp.java:17)
Jan 28, 2024 12:24:44 PM org.corp.TestApp main
INFO: Finish operation
```

Since we rely on the default **ConsoleHandler** configuration, the default log level is **INFO**, so we see **SEVERE** and **INFO** log

messages here.

Let us explore how we can work date and time in Java.

Exploring the Date-Time APIs

Introduced in JDK 8, the Date-Time API is provided through the **`java.time`** package, a set of classes representing date and time values in different formats. Before the Date-Time API, we relied on the **`java.util.Date`** class. The new Date-Time API was introduced to fill the gaps from the previous **`java.util.Date`** implementation.

This section covers some of the main aspects of the Date-Time API, including date, time, date with time, and zoned date with times. Let us start by checking how the **`LocalDate`** class works.

LocalDate

We use the **`LocalDate`** class when we are only interested in the date representation formed by days, months, and years. You may be working with an application that provides range filtering capabilities where the time granularity is defined by days, excluding any time aspects related to hours or minutes. Such a use case could benefit from the **`LocalDate`** capabilities. The following is how we can create a **`LocalDate`** representing the current date:

```
System.out.println(LocalDate.now()); // 2024-01-28
```

When calling the **`now`** method, it returns the current date based on the default time zone where the JVM is running unless we enforce a default time zone using something like below:

```
TimeZone.setDefault(TimeZone.getTimeZone("Japan"));
```

If we do not enforce a default timezone through the application, Java relies on the timezone defined by the operating system where the JVM is running. For example, my computer clock is configured to use the **`"Europe/Berlin"`** timezone. If executed on my computer, the `now` method would return the **`"Europe/Berlin"`** timezone.

We can pass another timezone if we do not want to use the default one:

`System.out.println(LocalDate.now(ZoneId.of("Japan")));`
`// 2024-01-29`

We can specify a different timezone using the **`of`** factory method from the **`ZoneId`** class. In the Japan timezone, the date returned one day ahead of the previous example, where I used the default timezone provided by the operating system.

We can get a **`LocalDate`** by providing the year, month, and day:

`System.out.println(LocalDate.of(2024, Month.JANUARY, 28)); // 2024-01-28`

The **`LocalDate`** is composed based on the provided date components like year, month, and day. We achieve the same result by providing only the year and the day of the year:

`System.out.println(LocalDate.ofYearDay(2024, 28)); // 2024-01-28`

The day 28 of 2024 falls in January, resulting in a **`LocalDate`** with that month. We can also provide a string that can be parsed and transformed into a **`LocalDate`**:

`System.out.println(LocalDate.parse("2024-01-28")); // 2024-01-28`

The string above must follow the default date format, which expects the year, month, and day in this order. If we try to pass something like **`"2024-28-01"`**, we get **`DateTimeParseException`**. We can solve it by using a data formatter:

```
String myFormat = "yyyy-dd-MM";

DateTimeFormatter myDateFormatter =
DateTimeFormatter.ofPattern(myFormat);

System.out.println(LocalDate.parse("2024-28-01",
myDateFormatter)); // 2024-01-28
```

There is an overloaded **`parse`** method from the **`LocalDate`** class that accepts a **`DateTimeFormatter`** to allow proper parsing of different date format strings.

We can also increase or decrease the **`LocalDate`**:

```
System.out.println(LocalDate.of(2024,
Month.JANUARY, 28).plusDays(1)); // 2024-01-29
```

```
System.out.println(LocalDate.of(2024,
Month.JANUARY, 28).minusMonths(2)); // 2023-11-28

System.out.println(LocalDate.of(2024,
Month.JANUARY, 28).plusYears(1)); // 2025-01-28
```

It is important to note that when we use methods like **`plusDays`** or **`minusMonths`**, they do not change the current **`LocalDate`** instance but instead return a new instance with the changed date.

What if we are only interested in the time aspect of a given date? That is when the **`LocalTime`** comes into play to help us.

LocalTime

The way we manipulate **`LocalTime`** objects is similar to how we manipulate **`LocalDate`**. The **`LocalTime`** class provides some factory methods that allow the creation of **`LocalTime`** objects in different ways:

```
System.out.println(LocalTime.now()); //
21:33:08.740527179

System.out.println(LocalTime.now(ZoneId.of("Japan")));
// 05:33:08.740560750

System.out.println(LocalTime.of(20, 10, 5)); //
20:10:05

System.out.println(LocalTime.of(20, 10, 5, 10));
// 20:10:05.000000010

System.out.println(LocalTime.parse("20:10:05"));
// 20:10:05
```

As with **`LocalDate`**, we can create **`LocalTime`** instances following the principle where we can get the current time with the **`now`** method or get a specific time by providing the time components like hour, minute, second, and nanosecond. It is also possible to parse a **`String`** representing time. Increasing or decreasing a **`LocalTime`** is also supported:

```
System.out.println(LocalTime.of(20, 10,
5).plusHours(5)); // 01:10:05

System.out.println(LocalTime.of(20, 10,
5).minusSeconds(30)); // 20:09:35
```

Let us check now how we can have an object that expresses both date and time.

LocalDateTime

By joining the date and time into a single object, the **`LocalDateTime`** provides a complete local representation of date and time. The term local means that these date-time objects do not contain any reference to their timezone, that is only possible with its variant with timezone support, which will be seen later in this section.

The following is how we can create a **`LocalDateTime`**:

```
System.out.println(LocalDateTime.now()); //
2024-01-28T22:21:38.049971256

System.out.println(LocalDateTime.now(ZoneId.of("Japan"))
// 2024-01-29T06:21:38.050009338

var localDate = LocalDate.of(2024, Month.JANUARY,
28);

var localTime = LocalTime.of(20, 10, 5);

System.out.println(LocalDateTime.of(localDate,
localTime)); // 2024-01-28T20:10:05

System.out.println(LocalDateTime.of(2024,
Month.JANUARY, 28, 20, 10, 5)); //
2024-01-28T20:10:05
```

As with other date-time classes, **`LocalDateTime`** has a **`now`** method that returns the current date and time. When creating **`LocalDateTime`** with a specific date and time, we can provide **`LocalDate`** and **`LocalTime`** objects to the factory method. We see next the date-time type that supports timezones.

ZoneDateTime

In all the previous classes we have seen so far, **`LocalDate`**, **`LocalTime`**, and **`LocatDateTime`** all provide a date and time presentation without regard to a given time zone. There are use cases where the time zone is helpful as part of the date-time object. We can save date-times using the time zone from the region where the Java application is running, like the scenarios where a system is

available for users in different time zones. We can also ensure the application always uses the **Coordinated Universal Time** (**UTC**), a universal way to represent time, no matter the region where the Java application is running. The use cases can be numerous, and to enable them, we have the **`ZoneDateTime`** class, which works similarly to the **`LocalDateTime`** but with support for time zones. The following is how to create a **`ZoneDateTime`** object representing the current date and time:

```
System.out.println(ZonedDateTime.now()); //
2024-01-28T23:46:26.709136750+01:00[Europe/Berlin]

System.out.println(ZonedDateTime.now(ZoneId.of("Japan"))
// 2024-01-29T07:47:33.206255659+09:00[Japan]
```

In addition to the date and time components, we have the time zone component as +01:00[Europe/Berlin] when calling the **`now`** method without a parameter defining a time zone and +09:00[Japan] when calling the **`now`** method with the Japan time zone parameter.

It is possible to create the **`ZoneDateTime`** from a **`LocalDateTime`**:

```
var localDateTime = LocalDateTime.now();

System.out.println(ZonedDateTime.of(localDateTime,
ZoneId.of("UTC"))); //
2024-01-28T23:55:27.406187953Z[UTC]
```

Remember that the **`now`** method gets the current date and time based on the operating system's default time zone unless we force the **Java Virtual Machine** (**JVM**) to use another time zone. The example above gets the local date and time and then converts it to UTC when creating a **`ZonedDateTime`**.

Instant

In the previous section, we saw how to use **`ZoneDateTime`** to create a moment representation with UTC. UTC stands for Coordinated Universal Time and is beneficial for situations where we want to represent time regardless of the geographical location or different time zones. Database systems usually adopt UTC as the standard for storing date-time values.

Java has the **`Instant`** class representing a moment with a

nanoseconds number counting from the epoch of the beginning of 1970 in UTC. Here is how we can create an **`Instant`**:

```
System.out.println(Instant.now()); //
2024-01-28T23:24:36.093286873Z

System.out.println(Instant.now().getNano()); //
93372648
```

Using **`Instant.now()`** is common when persisting Java data objects into database systems because it provides a UTC moment representation that is usually compatible with timestamp data types from most database technologies.

Functional programming with streams and lambdas

The cornerstone of **object-oriented programming** (**OOP**) is the idea of objects having states and data. For example, we may trigger an object's method to change its state. We can also make an object take the form of another object. The ability to transform objects' constitutes a mutation characteristic common with OOP languages. Employing this mutation can bring benefits in terms of flexibility because we can adapt and reuse the same object in different contexts. On the other hand, the mutation can bring problems because the object can change in unexpected ways, causing side effects in our system.

The Java architects introduced functional programming elements to the Java language to address the possible side effects caused by mutating objects and to allow a more declarative and concise way to develop software. Starting on Java 8, we have the **`Stream`** interface, which enables us to process a collection of objects in a functional way. There is also a functional-style syntax called lambda expression, which allows passing blocks of code, for example, as a parameter for a method or assigned to a variable. We also have functional interfaces that lambda expressions can implement. The Java API provides built-in helpful functional interfaces, but we can also implement our own. Let us start our exploration with lambda expressions.

Functional interfaces and lambda expressions

We consider an interface functional if it has one and only one

abstract method. We usually implement ordinary interfaces with a concrete class, but we can implement functional interfaces using lambda expressions. The Java API provides many built-in functional interfaces for different use cases. Unless you have a specific need, you will likely use already available functional interfaces. Let us look at some of these interfaces, starting with the **`Predicate`**.

Predicates

We use predicates to create expressions that always return a Boolean. The following is how the **`Predicate`** interface is defined in the Java API:

```
@FunctionalInterface
public interface Predicate<T> {
    // Code omitted
    boolean test(T t);
    // Code omitted
}
```

The **`@FunctionalInterface`** annotation is recommended but not mandatory. We can use this annotation to ensure the functional interface has only one abstract method. `T` is the generic parameter type used in the expression we create, and **`boolean`** is what we always return as a result of evaluating the expression. Look how we can implement a **`Predicate`** with a Lambda expression:

```
Predicate<String> emptyPredicate =  value -> value != null && value.isEmpty();
```

A lambda expression is composed of zero, one or more parameters on the left side, then an arrow, and finally, the expression body on the right side. Notice we are assigning this expression to a variable named **`emptyPredicate`**. There is no need to specify the value parameter type here because the Java compiler infers it from the type we provide on **`Predicate<String>`**. We can evaluate our lambda expression by calling the **`test`** method with different values, as follows:

```
System.out.println(emptyPredicate.test(null)); // false
```

```
System.out.println(emptyPredicate.test("")); //
true
```

```
System.out.println(emptyPredicate.test("abc")); //
false
```

Passing **`null`** or **`"abc"`** makes our expression return **`false`**. It returns **`true`** when we pass an empty **`String`**.

Functions

We use functions when we usually want to pass a value, do something with it, and return a result. That is how the **`Function`** functional interface is defined in the Java API:

```
@FunctionalInterface
public interface Function<T, R> {
   // Code omitted
    R apply(T t);
   // Code omitted
}
```

`R` is the result type we expect to get, and **`T`** is the parameter type we use inside our expression.

You may be wonder what **`R`** and **`T`** means. These letters are identifiers we use to define generic type parameters used by generic classes or interfaces. Function is a generic functional interface that receives two generic type parameters: **`R`** and **`T`**. Where **`R`** stands for return and **`T`** stands for type. Generics give us flexibility by providing a way to design classes and interfaces capable of working with different types.

The following is how we can declare and use the **`Function`** interface:

```
Function<Integer, String>
putNumberBetweenParentheses = (input) -> "("+input
+")";
```

```
var java = putNumberBetweenParentheses.apply(25);
```

```
var function =
putNumberBetweenParentheses.apply(8);
```

```
System.out.println(java); // (25)

System.out.println(function); // (9)
```

In the example above, **`Integer`** is the type of the parameter we are passing, and **`String`** is the return type. Our function gets an **`Integer`**, converts it to a **`String`**, and wraps it into parentheses.

Suppliers

There are scenarios where we need to get an object without providing any input. Suppliers can help us because that is how they behave: we call them, and they just return something. Following is how the Java API defines the **`Supplier`** interface:

```
@FunctionalInterface

public interface Supplier<T> {

  // Code omitted);

  T get();

}
```

`T` is the return type of the object we receive when calling the **`get`** method. Here is an example showing how to use a **`Supplier`**:

```
Supplier<List<String>> optionsSupplier = () ->
List.of("Option 1", "Option 2", "Option 3");

System.out.println(optionsSupplier.get()); //
[Option 1, Option 2, Option 3]
```

Notice that the parameter side **`()`** of the lambda expression is empty. If the functional interface does not expect any parameter, we do not provide it when creating the lambda expression.

Consumers

With **`Suppliers`**, we get an object without providing any input. Consumers work in the opposite direction because we give input to them, but they return nothing. We use them to perform some action in the provided input. The following is the **`Consumer`** functional interface definition:

```
@FunctionalInterface

public interface Consumer<T> {
```

```
    // Code omitted
    void accept(T t);
    // Code omitted
}
```

We pass a generic **`T`** type to the **`accept`** method that returns nothing. Here, we can see a lambda expression showing how to use the **`Consumer`** functional interface:

```
Consumer<List<String>> print = list -> {
for(String item : list) {
 System.out.println(item+": "+item.length());
}
};
print.accept(List.of("Item A", "Item AB", "Item ABC"));
```

Implementing the **`Consumer`** interface, we have a lambda expression that receives a **`List<String>`** as a parameter. This lambda shows that an expression can be a whole code block with multiple statements between curly brackets. Here, we are iterating over the list of items and printing their value and length. Note that this lambda expression does not return anything; it just gets the list and performs some action with it.

The Java API provides many other functional interfaces. Here, we have just covered some of the main interfaces to understand their fundamental idea and how they can help us develop code in a functional way.

Let us explore how to use streams to handle collections of objects.

Streams

Introduced together with functional interfaces and lambda expressions in Java 8, the **`Stream`** interface allows us to handle collections of objects functionally. One significant difference between streams and the data structures provided by the **`Collection`** interface is that streams are lazy loaded. For example, when we create a list of objects, the entire list is allocated in

memory. If we deal with large amounts of data, our application may struggle and have memory issues loading all objects into a list. On the other hand, streams allow us to process one object at a time in a pipeline of steps called intermediate and terminal operations. Instead of using a collection and loading everything in memory, we rely on a stream to process object by object, performing the required operations on them.

It is a common situation where getting results from a large database table is necessary. Instead of returning the results as a **`List`**, we can return them as a **`Stream`** to prevent memory bottlenecks in the application.

Streams may be beneficial for preventing memory bottlenecks and also for helping us handle collections of objects more flexibly. To understand how we can tap into the benefits provided by streams, we first need to understand the stream structure.

Streams are designed like a pipeline where we have the following components:

- Stream source
- Intermediate operations
- Terminal operation

The stream source is where we get the data we want to process as a stream. We can use a **`List`**, for example, to source a stream. Intermediate operations allow us to manipulate stream data in a pipeline fashion, where the output from one intermediate operation can be used as the input to another intermediate operation. As the last component, we have the terminal operation that determines the stream's final outcome. Terminal operations can return an object or handle the stream data without returning something.

Let us start by checking how we can source a stream.

Sourcing streams

The simplest way to initialize a stream is through the **`of`** factory method from the **`Stream`** interface:

```
Stream<String> streamOfStrings = Stream.of("Element
1", "Element 2", "Element 3");
```

A more recurrent use is creating streams out of an existing

collection like a **`List`**, for example:

```
var listOfElements = List.of("Element 1", "Element
2", "Element 3");

Stream<String> streamOfStrings =
listOfElements.stream();
```

Once we have a stream, the next step is to handle its data with intermediate and terminal operations. Let us first check how intermediate operations work.

Intermediate operation

For those familiar with Unix/Linux shells, you may know how helpful it is to use the output of a command as the input for another command through the usage of the pipe | character. The Java streams' intermediate operations work similarly by letting us use the result from one intermediate operation as the input for the next intermediate operation. Because of this continuing and compounding nature, these operations are called intermediate. They do not represent the final outcome of the stream processing.

Here are some common intermediate operations examples:

```
Stream<String> streamOfStrings =
Stream.of("Element 1", "Element 2", "Element 3");

Stream<String> intermediateStream1 =
streamOfStrings.map(value -> value.toUpperCase());

Stream<String> intermediateStream2 =
streamOfStrings.filter(value ->
value.contains("1"));

Stream<String> intermediateStream3 =
streamOfStrings.map(value ->
value.toUpperCase()).filter(value ->
value.contains("1"));
```

We start by creating a stream of strings. The first intermediate operation puts all string characters in uppercase. Note that intermediate operations always return a **`Stream`** object, so we can use it to perform another intermediate operation or terminate the stream processing with a terminal operation. Next, the second intermediate operation filters the stream data, returning only the

objects containing the **`"1"`** string. Finally, we concatenate the **`map`** and **`filter`** intermediate operations in the third intermediate operation.

The **`map`** intermediate operation receives a lambda expression implementing the **`Function`** interface as its parameter. Remember, a **`Function`** is a functional interface that accepts a parameter and returns an object. We are passing a **`String`** and returning another **`String`** with its characters in uppercase.

We are passing a lambda expression for the **`filter`** intermediate operation that implements the **`Predicate`** functional interface.

Instead of creating multiple streams containing intermediate operations, we can create one single stream with multiple intermediate streams, as shown follows:

```
streamOfStrings
    .map(value -> value.toUpperCase())
    .filter(value -> value.contains("1"))
    .filter(value -> value.contains("3"));
```

It is important to understand that we are not processing data by creating intermediate operations. The real processing only happens when we call a terminal operation. Let us check next if it works.

Terminal operation

Creating one or more intermediate operations on top of a stream does not mean the application is doing something with the stream data. We are only stating how the data should be processed at the intermediate level. The stream data is only processed after we call a terminal operation. Here are some examples of terminal operations:

```
Stream<String> streamOfStrings =
Stream.of("Element 1", "Element 2", "Element 3");
streamOfStrings
        .map(String::toUpperCase)
        .filter(value -> value.contains("1"))
        .forEach(System.out::println); // ELEMENT
1
```

```
        //.collect(Collectors.toList()); //
[ELEMENT 1]
        //.count(); // 1
```

After concatenating the **`map`** and **`filter`** intermediate operations, we applied a **`forEach`** terminal operation. Notice we use a more concise syntax instead of a lambda expression. We call this syntax method reference and can use it in scenarios where the parameter declared on the right side of the lambda expression is the same parameter used in the method called within the body part of the lambda expression. Here, we have a method reference and a lambda expression that produce the same result:

```
.forEach(System.out::println);
.forEach(value -> System.out.println(value));
```

In addition to **`forEach`**, we also have a **`collect`** terminal operation, which allows us to return the stream processing result as a **`List`**, for example. The **`count`** terminal operation counts the number of objects the stream still has.

Compiling and running the sample project

This chapter provides a sample project that applies the Java API concepts we have seen. The project is a Java application called **Report Generator**, which generates and saves reports in a text file. You can find the project's source at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2001**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

To compile the project, go to the *Chapter 1* directory from the book's repository. From there, you need to execute the following command:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the command below:

```
$ java -jar target/chapter01-1.0-SNAPSHOT.jar
Jan 30, 2024 9:44:15 PM
```

```
dev.davivieira.report.service.ReportService
generateReport

INFO: Starting to generate report

Jan 30, 2024 9:44:15 PM
dev.davivieira.report.service.ReportService
generateReport

INFO: Report generated with success
```

Conclusion

This chapter explored some of the Java APIs often used in most Java projects. Starting with the Java Collections Framework, we learned to use lists, sorts, and maps to provide a solid data structure foundation for Java applications. We saw how the NIO2 lets us easily manipulate files and directories. Keeping an eye on the possibility that a Java application cannot behave as expected, we explored creating, throwing, and handling exceptions. To ensure better visibility of what a Java application is doing, we learned how the Logging API helps to log system behaviors. Next, we explored how the Date-Time gracefully lets us handle different time and date components, including time zones. Tapping into functional programming, we learned how powerful stream and lambdas can be in developing code in a functional way.

This chapter reminded us of some cool things the Java API provides. The next chapter will dive deep into what is new in Java, especially the changes introduced between the Java 17 and 21 releases.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2
Exploring Modern Java Features

Introduction

A fascinating aspect of Java is that after more than twenty-five years since its first release, the language keeps innovating with new features and enhancing how we develop software. Many of us rely on Java for its robustness and commitment to staying as stable as possible regarding backward compatibility from new to old releases. An amazing point about Java is its capacity to be rock-solid for mission-critical applications and, simultaneously, a language offering cutting-edge features for applications that explore new concepts and ways of doing things.

As Java is a language that changes and evolves quickly, keeping up with every new feature may be challenging. So, in this chapter, we explore essential features introduced between the Java 17 and 21 releases that help us develop applications more efficiently.

Structure

The chapter covers the following topics:

- Getting more control over inheritance with sealed classes
- Increasing code readability with pattern matching
- Increasing application throughput with virtual threads
- Compiling and running the sample project

Objectives

By the end of this chapter, you can tap into Java's modern features to develop software more efficiently. By understanding virtual threads, for example, you will have the means to simplify the development and increase the throughput of concurrent systems. By employing pattern matching, and record patterns, you can make your code concise and easier to understand. With sealed classes, you will have fine-grained inheritance control and enhanced encapsulation.

The features and techniques presented in this chapter can increase the quality and efficiency of the code written in your Java projects.

Getting more control over inheritance with sealed classes

Java provides inheritance capabilities as an object-oriented language, allowing us to inherit classes and interfaces. When dealing with classes, inheritance occurs when extending a concrete or abstract class. With interfaces, we can extend or implement them. Whether we were dealing with classes or interfaces, until the Java 15 release (as a preview feature), there was no way to restrict who could inherit a class or interface. Letting anyone implement a class, for example, could cause issues in the application architecture.

Consider the scenario where we have an abstract class called **`Card`** to represent a physical card people use to access the facilities of a building. Possible implementations of the **`Card`** abstract class could be the **`PermanentCard`** or **`TemporaryCard`** classes with different rules regarding authorization to enter the building. Suppose now the system responsible for handling those cards will also support storing and using credit card information to charge people who want to use the building parking space. To allow it, a developer, for some reason, decides to create the **`CreditCard`** class by extending

the **`Card`** abstract class. The issue here is that the developer who created the **`Card`** abstract class designed it in a way with methods and attributes that would make sense only in the context of cards as a means for access control, not payment. The **`CreditCard`** class may end up inheriting from the **`Card`** abstract class the methods and attributes that are not supposed to be used in the context of payments but as means for access control.

In the scenario described, a developer can provide dummy implementations of abstract methods to fulfill the abstract class contract. The code and the feature delivered would work as expected, but we would have a design incongruence in our code. Sealed classes can help prevent such incongruence by enforcing which classes can implement the **`Card`** abstract class provided in our example.

The developer providing the credit card feature would look at the card-sealed abstract class and see that **`PermanentCard`** and **`TemporaryCard`** are the only possible allowed implementations.

Let us explore the scenario described above by learning how sealed classes can help us enforce our inheritance expectations.

Enforcing inheritance expectations

If you have been developing Java code for a while, you may have heard of the advantages of code reuse, which means avoiding rewriting logic that already exists somewhere in the codebase. On Java, the Has-A and Is-A class relationship establishes how an object can increase its capabilities. With the Has-A relationship, also known as composition, we can make a class acquire all the capabilities of another class by just referring to it, like in the following example:

```
class Printer {
    void print() {
        // Print something
    }
}
public class Report {
```

```
    private Printer printer;

     Report(Printer printer)  {
         this.printer = printer;
     }
     public void printSomething() {
         printer.print();
     }
}
```

The **`Report`** class has the **`Printer`** class as an attribute, which means all the visible attributes and methods from the **`Printer`** class will be available to the **`Report`** class through the printer attribute.

On the other hand, with the Is-A, we can rely on the inheritance between classes to extend the behaviors and data of another class. The code is as follows:

```
class Printer {
    protected void print() {
        // Print something
    }
}
public class Report extends Printer {
    public void printSomething() {
        print();
    }
}
```

We can access the inherited **`print`** method by extending the **`Printer`** into the **`Report`** class. From an object-oriented design perspective, it is counterintuitive that a **`Report`** is also a **`Printer`**. Still, the example above shows we can do something like that if we

want to access a method from another class through inheritance. So, to help prevent counterintuitive inheritances, we can rely on the Has-A relationship in favor of the Is-A because of the flexibility in composing class capabilities by referring to other classes instead of tightly coupling them through the Is-A relationship.

Still, there can be scenarios where the Is-A relationship is entirely valid to extend class capabilities and better represent the class design of the problem domain we are dealing with. A common use case is when we create an **`abstract`** class with the intent to provide some data and behavior that will be shared across other classes, like in the following example:

```
abstract class Report {
    String name;
    abstract void print();
    abstract void generate();
    void printReportName() {
        System.out.println("Report "+name);
    }
}
```

Based on the generic abstract **`Report`** class, we can provide concrete classes like **`PDFReport`** and **`WordReport`** containing specific logic to generate and print reports:

```
class PDFReport extends Report {
    @Override
    void print() {
        printReportName();
        System.out.println("Print PDF");
    }
    @Override
    void generate() {
        // Generate PDF
    }
```

```
}
class WordReport extends Report {
    @Override
    void print() {
        printReportName();
        System.out.println("Print Word");
    }
    @Override
    void generate() {
        // Generate Word
    }
}
```

Generating and printing reports are behaviors we can always expect from **`Report`** type objects, whether they are **`PDFReport`** or **`WordReport`** subtypes. However, what if someone extended the **`Report`** class to create, for example, the **`ExcelReport`** class with slightly different behavior:

```
class ExcelReport extends Report {
    @Override
    void print() { // Dummy implementation }

    @Override
    void generate() {
        // Generates Excel
    }
}
```

The **`ExcelReport`** class is only concerned with generating reports; it does not need to print anything, so it just provides a dummy implementation of the print method. That is not how we expect classes implementing **`Report`** to behave. Our expectation is based on the behaviors provided by the **`PDFReport`** and **`WordReport`**

classes. Java allows us to enforce such expectations with sealed classes. The code block below shows a sealed class example:

```
sealed abstract class Report permits PDFReport,
WordReport {
    String name;
    abstract void print();
    abstract void generate();
    void printReportName() {
        System.out.println("Report "+name);
    }
}
```

We know that the **`PDFReport`** and **`WordReport`** classes fulfill the **`Report`** abstract class contract by providing meaningful implementations of the **`print`** and **`generate`** abstract methods, so we seal the **`Report`** abstract class to restrict its inheritance to the classes we trust.

The classes implementing the sealed abstract **`Report`** class can be defined as **`final`**, **`sealed`**, or **`non-sealed`**. The following is how we can define it as **`final`**:

```
final class WordReport extends Report {
    // Code omitted //
}
```

We use the **`final`** keyword to ensure the hierarchy ends on the **`WordReport`** without further extension. It is also possible to define the child class as a **`sealed`** one, like in the following example:

```
sealed class WordReport extends Report permits
Word97, Word2003 {
    // Code omitted //
}
final class Word97 extends WordReport {
    // Code omitted //
}
```

```
final class Word2003 extends WordReport {
    // Code omitted //
}
```

The **`WordReport`** class is extensible only to **`Word97`** and **`Word2003`** classes. However, what if you wanted to make **`WordReport`** extensible to any class without restriction? We can achieve that using a **`non-sealed`** class like in the following example:

```
public sealed abstract class Report permits
WordReport {
    // Code omitted //
}
non-sealed class WordReport extends Report {
    // Code omitted //
}
```

There are no **`permits`** on the **`WordReport`** class because we want to make it extendable by any other class.

Another benefit of using sealed classes is that **`switch`** statements become more straightforward because the compiler can infer all the possible types of a sealed hierarchy, which removes the necessity to define the default case in a **`switch`** statement. Consider the following example:

```
sealed abstract class Report permits WordReport,
PDFReport {
    // Code omitted //
}
final class WordReport extends Report {
    // Code omitted //
}
final class PDFReport extends Report {
    // Code omitted //
}
```

```
class TestSwitch {
    int result(Report report) {
        return switch (report) {    // pattern
matching switch
            case WordReport wordReport -> 1;
            case PDFReport pdfReport   -> 2;
            // there is no need for default
        };
    }
}
```

The pattern matching **`switch`** receives the sealed abstract **`Report`** class as a parameter that permits only the **`WordReport`** and **`PDFReport`** classes. Because the Java compiler knows these two classes are the only possible types deriving from the **`Report`** class, we do not need to define a default case in the **`switch`** statement.

Sealed classes help us emphasize which class hierarchies should be controlled to avoid code inconsistencies due to unadvised inheritances.

Moving out from sealed classes, let us see how we can write more concise and easily understood code by applying pattern-matching techniques.

Increasing code readability with pattern matching

Introduced in the Java 17 release (it appeared initially as a preview feature on Java 14), pattern matching represents a notable effort to decrease boilerplate from the Java language. This feature identifies areas in the language where pattern-matching techniques make sense, allowing developers to write cleaner and simpler code. Let us start our exploration by understanding what pattern matching is, then where and how Java is applying it to improve the language.

Introduction to pattern matching

If you have ever faced the troubleshooting problem of searching for log messages containing a specific term, you may have used pattern-

matching techniques. In Unix-based environments, tools like sed or grep allow us to match a word term in a text file. For example, suppose we have a file called **`lines.txt`** with the following content:

```
Line 1 – Message A

Line 2 – Message B

Line 3 – Message C
```

Then we execute the following command:

```
$ grep "Line 2" lines.txt

Line 2 – Message B
```

Only the line containing the term **`Line 2`** is displayed.

The example above demonstrates the fundamental pattern-matching mechanism of checking a value against a pattern. The value is the content from the **`lines.txt`** file, and the pattern is *Line 2,* which matches *Line 2—Message B,* representing the result of the pattern-matching evaluation. The whole idea behind pattern matching is based on the following components:

- Matching target
- Pattern expression
- Result

The matching target is the element we are matching against. In the previous examples, we used a text file and a multi-line string, but we can use other things, as we will see soon. The pattern expression represents what we are looking for in the matching target, and the result is the output captured when we apply the pattern expression against the matching target.

Java has introduced pattern matching for the following language features:

- Checking if an object is of a given type (**`instanceOf`**)
- Switch statements
- Record classes

Let us start checking the advantages of pattern matching for checking the object type.

Pattern matching for type

A typical Java construct is when we need to check if an object is of a given type, and if the type check matches, we then perform type casting and do something with the converted type. The following example illustrates it:

```
Object object = "Java";

if(object instanceof String)  {

String name = (String) object;

System.out.println(name+" length is
"+name.length()); // Java length is 4

}
```

The **`if`** conditional checks whether the **`object`** variable is of type **`String`**. If it is true, it casts the object to a **`String`**, assigns it to the **`name`** variable, and finally prints it. The following code shows we can achieve the same result using pattern matching, as follows:

```
Object object = "Java";

if(object instanceof String name) {

System.out.println(name+" length is
"+name.length()); // Java length is 4

}
```

Note that the **`name`** variable is part of the **`instanceOf`** expression. We call **`name`** a pattern variable that we can use if the pattern expression is evaluated as true. The object variable containing the **`"Java"`** string represents the matching target, and the **`String`** class type (after the **`instanceOf`**) represents the type pattern we match against. The **`name`** pattern variable is the result of storing the object converted to a **`String`** so we can use it in the if expression body.

We can use pattern matching to simplify the **`equals`** method. Consider the **`equals`** implementation for the **`Person`** class:

```
public class Person {

    private String name;

    public boolean equals(Object object) {

        if (!(object instanceof Person)) {
```

```
            return false;
        }
        Person person = (Person) object;
        return name.equals(person.name);
    }
}
```

The previous example shows we can implement the **`equals`** method without pattern matching. The following is the pattern-matching version:

```
public class Person {
    private String name;
    public boolean equals(Object object) {
        return object instanceof Person person &&
person.name.equals(name);
    }
}
```

Pattern matching allows us to inline the **`equals`** implementation by removing unnecessary boilerplate code.

In addition to checking the class types, pattern matching can also be used with **`switch`** expressions. We will explore how to do so next.

Pattern matching for switch statement

The pattern matching for switch statements is a preview feature from the Java 19 release, released as a non-preview feature on Java 21. It allows us to leverage pattern matching while evaluating case conditions. To understand how it works, let us consider an example where we use **`if-else`** instead of switch **`switch`**:

```
public class SwitchPatternMatching {
    public static void main(String... args) {
        checkParamType(1); // The param 1 is an
Integer
    }
```

```
    private static void checkParamType(Object
param) {
        if (param instanceof String s) {
            System.out.println("The param " + s +
" is a String");
        } else if (param instanceof Integer i) {
            System.out.println("The param " + i +
" is an Integer");
        } else if (param instanceof Double d) {
            System.out.println("The param " + d +
" is a Double");
        } else {
            throw new
IllegalStateException("Unexpected value: " +
param);
        }
    }
}
```

The method **`checkParamType`** uses pattern matching to check if the passed **`Object`** type is a **`String`**, **`Integer`**, or **`Double`**. When using pattern matching with **`switch`** statements, instead of dealing with values, we deal with types, as shown in the following example:

```
private static void checkParamType(Object param) {
switch(param) {
 case String s -> System.out.println("The param
"+s+" is a String");
 case Integer i -> System.out.println("The param
"+i+" is an Integer");
 case Double d -> System.out.println("The param
"+d+" is a Double");
 default -> throw new
IllegalStateException("Unexpected value: " +
```

```
param);
}
}
```

On the left side of the **->** operator, we have the types **String**, **Integer**, and **Double** followed by their pattern variables **s**, **i**, and **d** respectively. On the right side of the **->** operator, we use the pattern variable to print a message saying which type is the **param** object. It is also possible to include a **boolean** expression together with the type checking we are doing when evaluating the case:

```
private static void checkParamType(Object param) {
switch(param) {
                // Code omitted
 case Integer i when i>0 ->
System.out.println("The param "+i+" is a positive
Integer");
 // Code omitted;
}
}
```

Here, we check if the **param** is of type **Integer**. Next, we use the **when** clause to check if **i** is a positive **Integer**. The right side of the case **->** operator is only executed if the previous two checks pass. We call guarded patterns the usage of the **when** clause with the case label.

Pattern matching for record

Records are immutable classes we can use as data carriers in Java. A record class provides default **hashCode**, **equals**, and attribute accessor (e.g., **title()**, **author()**, and **year()**) methods. In the Java 19 release, the pattern match feature was extended to allow data extraction from record classes. Consider the following record class:

```
record Book(String title, String author, int year) {
}
```

A record class automatically implements accessor methods like

`title()` or **`author()`**, returning their respective values. As automatically generated accessor methods are something a record class will always provide, the pattern matching for records leverages it to allow a faster way to extract record attribute data, as shown in the following example code:

```
record Book(String title, String author, int year)
{ }

public class RecordPatternMatching {

    public static void main(String... args){

        var book = new Book("Book Title", "John
Doe", 1996);

        printAuthor(book);

        var bookWithNullTitle = new Book(null,
"John Doe", 1996);


System.out.println(getTitle(bookWithNullTitle));
// Unknown

    }

    public static void printAuthor(Object object)
{

        if (object instanceof Book(var title, var
author, var year)) {

            System.out.println(author); // John
Doe

        }

    }

    public static String getTitle(Object object) {

        return switch (object) {

            case Book(var title, var author, var
year) when title !=null -> title;

            default -> "Unknown";

        };
```

```
    }
}
```

We create a new **Book** record and pass it to the **printAuthor** method. Then, we extract the book's author attribute value with the following construct:

```
if (object instanceof Book(var title, var author,
var year)) {
System.out.println(author); // Book Title
}
```

The pattern type we want to match, what comes after the **instaceof** operator, is the **Book** record type. We declare it with all the required record's initialization parameters, including **title**, **author**, and **year**. Note we are using the **var** keyword to refer to the **Book** record attributes. It is legal to use it from a pattern record-matching expression. What follows inside the **if** block is that we print the book's author using the parameter we previously defined using the **var** keyword. Under the hood, Java is calling the accessor method **author** from the **Book** record class.

With the **getTitle(Object object)** method, we follow the same idea when defining the case label for the switch statement. There, we again declare the **Book** record type with its initialization parameters. We also use the guarded pattern to check if the title value is not **null**.

Increasing application throughput with virtual threads

A thread is always linked to a process in the operating system. Multiple threads in each process share the same resources because they are part of the same process. Some programs may contain just one thread because their logic does not require concurrent processing. In this case, you see a process and a thread in the operating system. Other programs may create multiple threads because they must trigger different sequences of activities that must be executed concurrently, such as one thread per request.

A server application may receive multiple requests at the same time. A single process in the operating system represents a running

application. To serve various requests simultaneously, the server application can allocate individual threads to serve each request. However, such an approach comes with a computing resource price. Threads are expensive to create because they need considerable memory and CPU. Bottlenecks may quickly happen if you create more threads than available computing resources. Careful usage of threads is one of the challenges developers face when building concurrent applications.

With these basic processes and thread concepts in mind, let us see how Java deals with threads.

Understanding Java platform threads

We know that Java code runs in the **Java Virtual Machine** (**JVM**), executed as a process in the operating system. Every Java program has at least one thread, called the main thread. When we create a platform thread in Java, the JVM triggers the creation of a physical thread in the operating system. It is called a platform thread because there is a direct link between the thread object in Java and the physical thread in the OS. The following figure shows how it works:

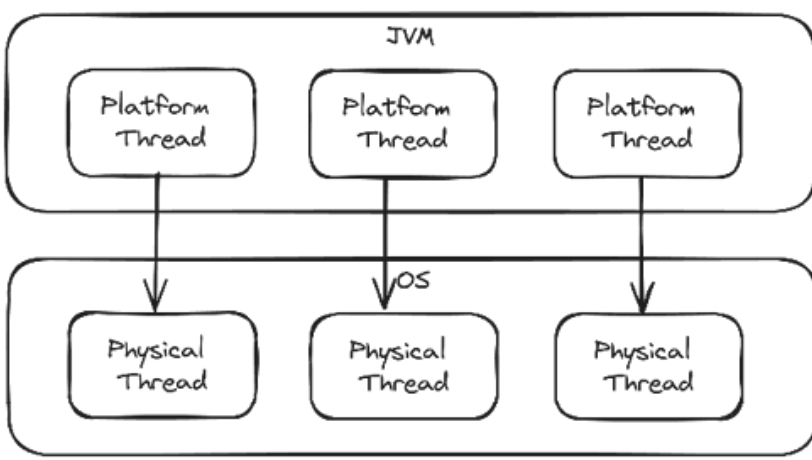

Figure 2.1: *The JVM platform and OS physical threads*

The new virtual thread implementation introduced the term platform thread to differentiate traditional (now called platform threads) from virtual threads.

Developing multi-thread applications using platform threads has been the standard approach since the first Java release. This approach worked well for quite a while, but as Java applications started to deal with more intense concurrent workloads, such an approach began to show some limitations. Let us assess these limitations.

Limitations of platform threads

The code developed using platform threads is based on the imperative programming paradigm. Imperative programming is how we develop software through sequential statements that change application state through code logic that relies on conditional expressions, and control flows like while and for loops. Imperative programming is predictable, straightforward to understand, and simple to debug. Most Java programs are written using imperative programming.

A critical challenge with imperative programming is when we need to deal with applications doing many blocking IO operations. We call blocking all system calls that need to wait for data from an external source, like when accessing a database or waiting for an API call to respond. In the upcoming section, we will assess how blocking IO operations can cause trouble and decrease the application's data processing throughput.

Platform threads and blocking IO operations

Suppose you have a back-end server application, an API, that is responsible for doing expensive calculations using data from a database. These calculations are processed inside a traditional platform thread. Consider now the scenario where the API receives 100 concurrent requests. It means that 100 threads must be provided to serve each request. Let us say we need 20 MB of RAM to create each thread. So, we will need at least 2 GB RAM to process all those requests. After creating a thread, we make a database call to perform the calculation and await a response. We say a thread is blocked when it is waiting for a response. The application will only continue execution once it gets a response or times out. Now, consider that every thread usually takes 20 seconds to get a response from the database, and during this interval, more additional requests arrive at the API. If no more than 2GB RAM is

available in the machine running our server application, we will not be able to create more threads because there is no more available memory. So, those additional requests will have to wait until the first requests finish and memory is released to allow the creation of new platform threads. The following figure describes the scenario:

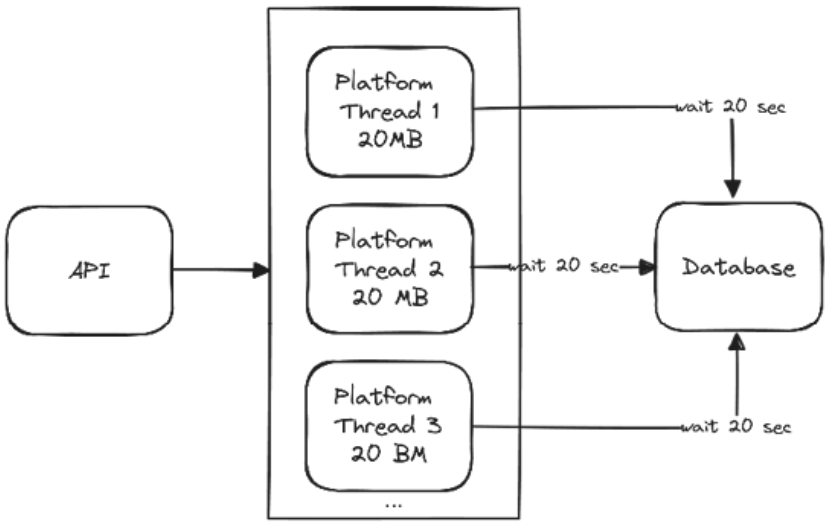

Figure 2.2: *Platform threads memory usage*

The issue here lies in the platform thread waiting for a response. The thread is consuming memory resources but is not doing anything; it is only waiting for a database response. We cannot process more requests because no free available memory exists to create more threads, decreasing our program throughput capacity.

Besides being expensive to build, platform threads take time to create. Creating a new thread object for every request is also a costly activity in terms of time. To overcome this issue, we can create thread pools with reusable threads. Having a thread pool solves the overhead problem of creating new threads, but we still have the hardware limitation on the number of total threads that can be created.

A technique involving reactive programming has emerged in Java to solve the throughput problem caused by blocking IO operations. Let us look at how it works.

Handling blocking IO with reactive programming

In the previous section, we discussed how imperative programming is more developer-friendly because of its sequential nature, which makes it more predictable and maintainable. Although easier to write, programs that rely on imperative programming may face bottlenecks when using platform threads to handle many concurrent IO-blocking operations.

As an alternative to imperative programming, we have reactive programming, where a program is designed to react to events. Instead of describing sequentially what an application must do, we need to think about which events can occur and how our application will respond to such events. Reactive programming is often used when an application needs to overcome blocking IO issues. An application can address such issues by running asynchronous tasks.

In the reactive programming approach, asynchronous behavior is usually achieved through non-blocking IO threads. These threads trigger IO operations and are immediately released instead of being blocked waiting for a response. A continuation mechanism on non-blocking IO threads allows them to receive a callback response once the IO operation is finished, allowing an IO thread to continue the task execution. The following figure illustrates how non-blocking IO threads deal with IO operations:

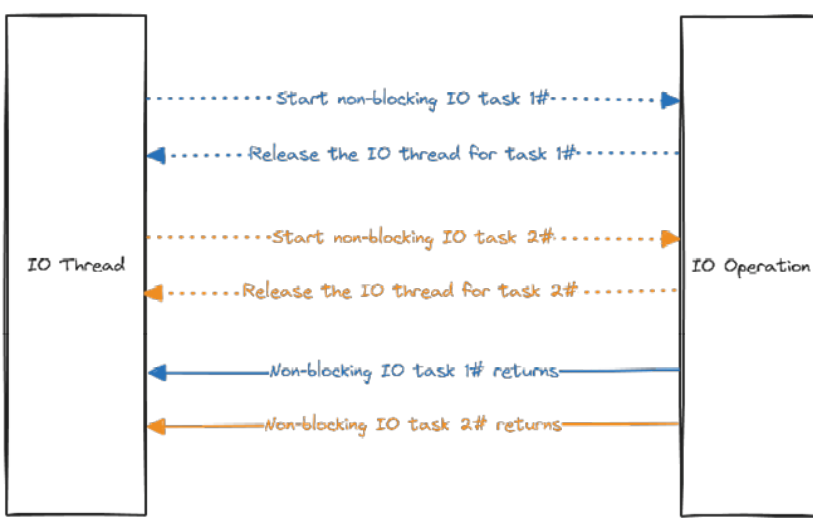
IO Thread
IO Operation
Start non-blocking IO task 1#
Release the IO thread for task 1#
Start non-blocking IO task 2#
Release the IO thread for task 2#
Non-blocking IO task 1# returns
Non-blocking IO task 2# returns

Figure 2.3: IO thread triggering IO operation

In the image above, the non-blocking IO tasks 1# and 2# run in parallel in non-blocking IO threads, which means no threads hanging waiting for a response. Once the IO operation finishes, the non-blocking IO task resumes and continues its execution. Once the IO operation finishes, the non-blocking IO task may continue in a different thread than it had started.

The fact that the execution of non-blocking IO tasks is distributed in different threads creates a debugging challenge because the task execution stack trace will be scattered among distinct threads. If something fails, we may need to investigate the execution of more than one thread to understand what happened. In imperative programming, for example, we can rest assured the stack trace of a single task is confined to one thread, which considerably decreases the debugging effort.

Reactive programming may be challenging not only on the debugging side. The reactive code also differs, relying heavily on functional programming constructs like streams and lambda expressions. Not all developers are well-versed in such a programming style.

Although reactive programming solves the IO blocking problem, it may create maintainability problems because it is more difficult to debug and write reactive code. Java architects aware of this situation came up with a solution that allows us to leverage non-blocking IO threads and write and debug code as we do in the imperative programming style. We call this solution virtual threads. Let us see how it works.

Writing simple concurrent code with virtual threads

We checked earlier that traditional platform threads are linked directly to an OS physical thread. Virtual threads, on the other hand, are connected to platform threads. Another critical point is that instead of the 1:1 relationship between platform and OS physical threads, we have a n:1 relationship between virtual and platform threads. In other words, we can have many virtual threads on top of a single platform thread. This is a fundamental characteristic of the virtual threads feature as a solution to overcome the thread bottleneck we have in the traditional platform

threads model. The following is an illustration of how virtual threads are arranged:

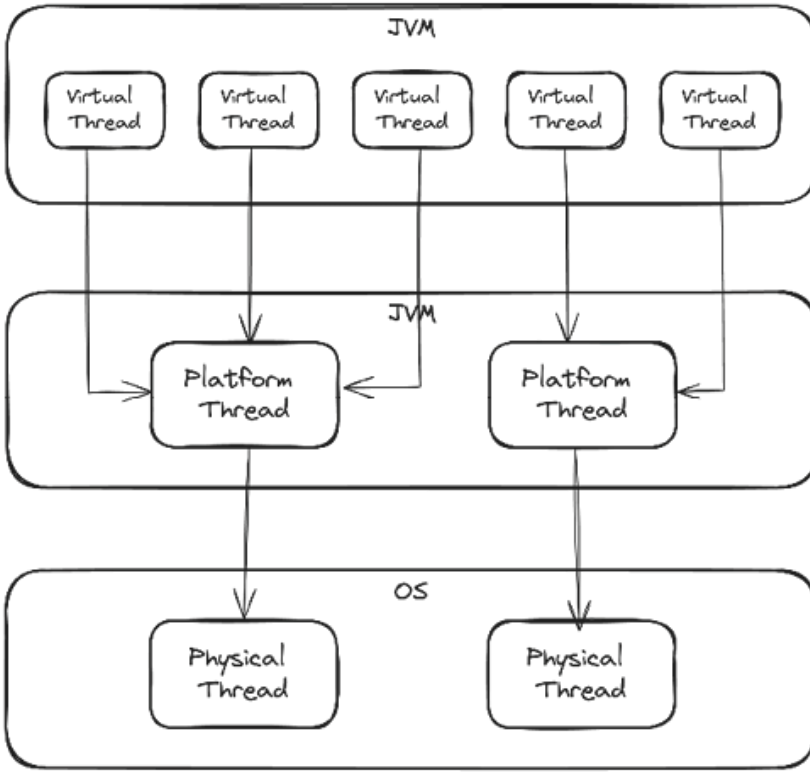

Figure 2.4: *Virtual and platform threads*

Virtual threads share the same platform thread, so they are cheap to create because creating them does not make the OS allocate more memory. How do they work under the hood?

Whenever an IO task is executed inside a virtual thread, such thread is suspended from the platform thread until the IO task finishes. The virtual thread runs until it is blocked, usually by an IO operation. When that happens, the virtual thread is removed from execution, and a new virtual thread can be selected to run in a platform thread. There is no guarantee that a virtual thread that was removed will run as soon as its blocking operation is completed, nor is there any guarantee that it will execute on the same platform thread.

Virtual threads allow us to write imperative synchronous code style and benefit, at the same time, from not blocking the platform thread from handling IO tasks from other virtual threads. To get an idea of the difference between reactive asynchronous code and imperative synchronous code, consider the following example that fetches data from an external source using an asynchronous programming style:

```
public class AsyncApp {

    public static void main(String... args) throws
ExecutionException, InterruptedException {

        CompletableFuture.supplyAsync(() ->
fetchURL().get()).thenAccept(AsyncApp::print).join();
// OK

    }

    static void print(String result) {

        System.out.println(result);

    }

    static Supplier<String> fetchURL() {

        return () -> {

            try {

                return getResponseMessage();

            } catch (Exception e) {

                throw new RuntimeException(e);

            }

        };

    }

    static String getResponseMessage() throws
Exception {

        var providedUrl = URI.create("https://
davivieira.dev").toURL();

        var con = (HttpURLConnection)
providedUrl.openConnection();

        con.setRequestMethod("GET");
```

```
        return con.getResponseMessage();
    }
}
```

The **`CompletableFuture`** is the Java class that lets us write asynchronous code. We pass the code we want to execute asynchronously to the **`supplyAsync`**. Then, after we get the result, we display it using the **`print`** method we passed to the **`accept`** method. Nothing will be printed if we do not call the **`join`** method because **`CompletableFuture`** triggers a non-blocking operation, which means the code will finish before we print the **`OK`** message. By calling **`join`**, we block the main thread until the **`CompletableFuture`** completes its task.

The following is how the same logic can be expressed using synchronous imperative programming style using a traditional platform thread:

```
public class SynchronousApp {
    public static void main(String... args) {
        Thread.ofPlatform().start(() ->
print(fetchURL().get())); // OK
    }
   // Code omitted
}
```

The code is slightly more straightforward but relies on a blocked platform thread until we finish execution. Now, consider the code using a virtual thread:

```
public class VirtualThreadApp {
    public static void main(String... args) throws
Exception {
        Thread.ofVirtual().start(() ->
print(fetchURL().get())); // Prints nothing
    }
   // Code omitted
}
```

Nothing is printed because when we start the IO network blocking call operation to fetch a URL, the JVM identifies it and automatically suspends the virtual thread for us. Since the virtual thread unblocks the IO operation, the program finishes before it gets a response. We can get the result of a virtual thread by calling the **`join`** method:

```
Thread.ofVirtual().start(() ->
print(fetchURL().get())).join(); // OK
```

By calling the **`join`** method in a virtual thread, we force the caller, in our example, the main thread, to wait until the virtual thread returns its result.

When we create a virtual thread, the JVM mounts it in a carrier thread, which is a platform thread. During the virtual thread execution, the JVM unmounts the virtual thread from the carrier thread if an IO operation is found. The virtual thread remains unmounted until the IO operations finish. After finishing it, the virtual thread is mounted again in the same carrier thread or a different one that continues the task execution.

By unmounting a virtual thread, the JVM releases the carrier thread so another virtual thread can mount it. Multiple virtual threads are mounted and unmounted in the carrier thread as their IO operations start and finish.

Instead of manually creating virtual threads, we can rely on the **`ExecutorService`** to create virtual threads for us. To accomplish it, we can use the static factory method **`newThreadPerTaskExecutor`** from the **`Executors`** class that returns an **`ExecutorService`**:

```
public class VirtualThreadApp {

    public static void main(String... args) throws
Exception {

        runTwoVirtualThreadsAtTheSameTime();

    }

    static void
runTwoVirtualThreadsAtTheSameTime() {

        final  virtualThreadfactory =
```

```
Thread.ofVirtual().factory();
        try (var executor =
Executors.newThreadPerTaskExecutor(virtualThreadfactory)
{
            executor.submit(() ->
print(fetchURL().get()));
            executor.submit(() ->
print(fetchURL().get()));
        }
    }
    static void print(String result) {
        System.out.println("Thread:
"+Thread.currentThread()+" - Result: "+result);
    }
   // Code omitted
}
```

We use the **`Thread.ofVirtual().factory()`** to build a virtual thread factory that is used at **`Executors.newThreadPerTaskExecutor(virtualThreadfactory)`** to create a new virtual thread per each task. In the example above, we submitted two tasks that are executed in two different virtual threads, producing the following output:

```
Thread: VirtualThread[#29]/runnable@ForkJoinPool-1-
worker-3 - Result: OK
Thread: VirtualThread[#31]/runnable@ForkJoinPool-1-
worker-2 - Result: OK
```

The virtual threads are executed in parallel, and their IO operations do not block the carrier (platform) thread they use. There are specific scenarios, though, where the virtual thread is not unmounted from the carrier thread. When this happens, we say the virtual thread is pinned to a carrier thread. Such a situation occurs when:

- We have a virtual thread executing code inside a synchronized block

- The virtual thread runs a native method or foreign function

When that happens, virtual threads prevent the platform thread from being mounted by other virtual threads. To overcome it, new platform threads need to be created to serve new virtual threads. So, caution is recommended when using virtual threads in one of the above scenarios. For scenarios where synchronized locks are necessary, we can consider replacing them with the **`ReentrantLock`**, which allows a thread to re-acquire a lock already held.

If you are working on a Java project or starting a new one where IO-intense operations are expected to occur frequently, consider using virtual threads to enhance the application performance.

Compiling and running the sample project

As a sample project, we have a Java application called Remote File Converter. It is a dummy converter that downloads files from the internet and converts them to different file types. This application illustrates the usage of sealed classes, pattern matching, string templates, and virtual threads.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2002**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

To compile the project, go to the *Chapter 2* directory from the book's repository. From there, you need to execute the following command:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the following command:

```
$ java -jar target/chapter02-1.0-SNAPSHOT.jar

Feb 25, 2024 3:04:58 AM
dev.davivieira.file.service.FileConverterService
convertFile

INFO: Downloading file from https://davivieira.dev/
```

```
file.pdf

Feb 25, 2024 3:05:00 AM
dev.davivieira.file.service.FileConverterService
convertFile

INFO: The file.pdf has been sucessfully downloaded

Feb 25, 2024 3:05:00 AM
dev.davivieira.file.service.FileConverterService
convertFile

INFO: Converting file.pdf to WORD

Feb 25, 2024 3:05:00 AM
dev.davivieira.file.service.FileConverterService
convertFile

INFO: Conversion to WORD was successfull!
```

The converted file is called `converted-from-pdf-to-word.docx`.

Conclusion

By keeping ourselves up-to-date with modern Java features, we can solve problems more efficiently by tapping into the newest features of the Java language. In this chapter, we had the chance to explore how sealed classes help us enforce inheritance expectations. We learned how pattern matching allows us to better deal with logic that does type casting using `instanceof`, including the `switch` statement and data extraction from record classes. We concluded by learning how virtual threads can significantly increase the throughput capacity of IO-intensive applications.

In the next chapter, we will explore the technologies and techniques we can use to handle relational databases in Java, such as the **Java Database Connectivity** (**JDBC**), which provides an interface that simplifies the interaction with databases, and the *Jakarta Persistence*, which lets us map Java classes to database tables.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 3
Handling Relational Databases with Java

Introduction

Relational databases continue to be widely used in Java projects that require data to be stored in a structured form. In the schema-enforced structure of relational databases, developers can store data in tables where each column has a specific data type. Java provides good support for those wanting to interact with relational databases. Serving as the foundation for database access and data handling, we have the **Java Database Connectivity** (**JDBC**) specification. On a higher level, we have the Jakarta Persistence (previously known as Java Persistence API), which allows us to map Java classes to database tables. Understanding how JDBC and Jakarta Persistence work is essential for anyone involved in Java projects that depend on relational databases.

Structure

The chapter covers the following topics:

- Introduction to JDBC

- Simplifying data handling with the Jakarta Persistence
- Exploring local development approaches when using databases
- Compiling and running the sample project

Objectives

By the end of this chapter, the reader will have the skills to develop Java applications that correctly employ the JDBC and Jakarta Persistence to deal with databases. The reader will also learn about the approaches to providing databases while locally running and developing a Java application.

Introduction to JDBC

The **Java Database Connectivity** (**JDBC**) specification defines how Java applications should interact with relational databases. It is also an API composed of interfaces describing how to connect and handle data from databases. Applications relying on the JDBC may benefit, to a certain extent, from the platform-independent nature of the JDBC specification, which enables applications to change database vendors without significant refactoring on the code responsible for connecting to the database. Changing database technologies is often a non-trivial activity that requires considerable refactoring, primarily due to the differences in the **Structured Query Language** (**SQL**) syntax and data types that may occur across different database vendors. That is a problem of database vendors employing non-ANSI-compliant usage of SQL, which can cause trouble for applications changing from database vendor A to B.

The JDBC specification provides standardization on an application level regarding the fundamental operations of any relational database. As fundamental operations, every relational database must support operations based on the following:

- **Data Definition Language**: **Data Definition Language** (**DDL**) operations encompass things like creating or changing a database table. It is the language that deals with the structure in which the data is organized in a relational

database.

- **Data Manipulation Language**: **Data Manipulation Language** (**DML**) operations, on the other hand, deal directly with data from a relational database. They provide a language for selecting, inserting, updating, and deleting data from a relational database.

Database vendors comply with the JDBC specification by implementing the JDBC API interfaces from the **`java.sql`** and **`javax.sql`** packages. These interfaces describe which methods can be used to, for example, create a connection to a database or send an SQL statement to select data from a table. Let us start our exploration by learning how to use Java to connect to a relational database.

Creating a database connection with the JDBC API

There are two ways to create a database connection: the first using the **`DriverManager`** class and the second using the **`DataSource`** interface. The second way is preferred because its reliance on the **`DataSource`** interface allows Java applications to create database connections without dealing with implementation details from the classes that implement the **`DataSource`** interface. A significant motivation to rely on the **`DataSource`** instead of the **`DriveManager`** is that with the latter, you would have to implement your connection pooling mechanism responsible for starting, closing, and caching database connections. With the **`DataSource`**, all such capabilities come for free, already implemented by the database vendor.

Getting a database connection with the DriverManager class

The **`DriverManager`** class has a method called **`getConnection`** that receives the database connection URL, credentials, and additional connection properties as parameters if required. That method returns a **`Connection`** object that can be used to start interacting with a database. The following is an example of how to create a database connection using the **`DriverManager`**:

```
public class JdbcConnection {

    public static void main(String... args) throws
```

```
Exception {
        var user = "test";
        var password = "test";
        var dbName = "test";
        var connection =
getConnectionWithDriverManager(user, password,
        dbName);
        System.out.println(connection.isClosed());
// false
    }
    public static Connection
getConnectionWithDriverManager(String user,
    String password, String dbName) throws
SQLException {
        var dbProvider = "mysql";
        var dbHost = "127.0.0.1";
        var dbPort = "3306";
        return
        DriverManager.getConnection(
        "jdbc:"+dbProvider+"://"+dbHost+":"+dbPort
+"\"+dbName);
    }
}
```

Assuming a MySQL Server is running locally with a user, password, and database name defined as **`test`**, we compose a database URL connection using the following structure:

`protocol:provider:host:port/database`

The protocol is usually **`jdbc`**. The provider represents the database vendor, which in our case is **`mysql`**. As the database is running on the same machine as the Java application, the host is **`127.0.0.1`**. The MySQL server from our example uses the default port, **`3306`**. The database name is **`test`**.

After getting the connection, we confirm if it is opened by calling the **`isClosed`** method, which returns **`false`**.

For this approach to work, we must ensure the JDBC driver for MySQL is loaded into the Java application's class or module path. The JDBC driver usually comes as a JAR file, which the database vendor provides.

The same **`Connection`** object can be acquired by using the **`DataSource`** interface, as shown in the upcoming section.

Getting a database connection with the DataSource interface

An application server may define the connection details, such as the database name, host, and credentials. The Java program running inside the application server does not need to specify the connection details, as it does when using the **`DriverManager`**. Instead, the Java program can rely on an alias name bound to the data source connection details from the application server. Next, we will see the steps for connecting to a database using the **`DataSource`** interface:

1. Configure and create a **`DataSource`** object:

```
private DataSource createDatasource(String user, Str
password, String dbName) {
    var dataSource = new MysqlDataSource();
    dataSource.setPort(3306);
    dataSource.setUser(user);
    dataSource.setPassword(password);
    dataSource.setDatabaseName(dbName);
    return dataSource;
}
```

   a. The application server would typically be responsible for creating the **`DataSource`** object, but we are creating it here to demonstrate how the **`DataSource`** configuration works. Note that we are using the **`MySQLDataSource`** class, which implements the **`DataSource`** interface. The

MySQL JDBC driver provides the **`MySQLDataSource`** class, which must be present in the application class or module path.

2. Create an **`InitialContext`** and bind a JNDI name to the **`DataSource`** object:

```
private Context createAndBindContext(DataSource data
throws NamingException {
    var env = new Hashtable(Map.of(Context.INITIAL_C
    "org.osjava.sj.SimpleContextFactory"));
    Context context = new InitialContext(env);
    context.bind("jdbc/testDB", dataSource);
    return context;
}
```

   a. The method above produces a **`Context`** object that can be used for database connection lookups based on the JNDI name. Like the **`DataSource`**, **`Context`** objects are usually provided by the Java program's application server. What we have here is a standalone **`Context`** provider to illustrate how **`Context`** creation works. The **Java Naming and Directory Interface** (**JNDI**) is a Java API that allows clients to look up or discover resources such as databases. With the JNDI, all that a client needs to know is the JNDI URI, which is resolved by the application server, which has resource details for a given JNDI.

   b. Observe that we assign to the **`env`** variable a **`Hashtable`** object containing a **`Map`** as its constructor parameter. This **`Map`** has the **`Context.INITIAL_CONTEXT_FACTORY`** as the key and the **`org.osjava.sj.SimpleContextFactory`** string as the value. Because we are not running the sample code in an application server—such as the Jboss or Weblogic, for example—we need to provide a context factory that will

produce a **`Context`** object for us. We can emulate the application server's responsibilities using the Simple-JNDI[1] library that produces **`Context`** objects. The Simple-JNDI is registered with the **`org.java.sj.SimpleContextFactory`** class. The **`Hashtable`** object is then passed as the constructor parameter for the **`InitialContext`** object. Finally, we bind the name **`jdbc/testDB`** to the **`DataSource`** object containing the database connection details.

3. Getting a connection using the **`DataSource`** object:

```
public static Connection getConnectionWithDataSource
String password, String dbName) throws Exception {
    var dataSource = createDatasource(user, password
    var context = createAndBindContext(dataSource);
    return
     ((DataSource)context.lookup("jdbc/testDB")).get
}
```

   a. This method combines what we did in the previous two steps. First, it creates a **`DataSource`** by calling **`createDatasource(user, password, dbName)`**. Second, it creates a **`Context`** and binds it to the **`DataSource`** created previously. The last line shows how the Java program performs a **`DataSource`** lookup by simply passing the JNDI name **`"jdbc/testDB"`** to the lookup method from the **`Context`** object. Calling the **`lookup`** method returns a **`DataSource`** object that we use to obtain a **`Connection`** object using the **`getConnection`** method.

4. We can use this **`Connection`** object to start interacting with the MySQL database:

```
public static void main(String... args) throws Excep
    var user = "test";
```

```
        var password = "test";
        var dbName = "test";
        var connection = getConnectionWithDataSource(use
              dbName);
        System.out.println(connection.isClosed()); // fa
    }
```

a. When working with production-grade Java projects, you will not need to manually create the **`DataSource`** and **`Context`** objects because they will be provided through the application server or the framework your Java program is using.

Now that we know how to set up a database connection using JDBC, let us explore how to send and process statements to the database.

Executing simple queries with the Statement

Before we start exploring the possibilities the **`Statement`** interface provides, let us consider the MySQL table created based on the following SQL code:

```
CREATE TABLE PERSON (
    ID int NOT NULL AUTO_INCREMENT,
    FIRST_NAME varchar(255),
    LAST_NAME varchar(255),
    AGE int,
    COUNTRY varchar(255),
    PRIMARY KEY (ID)
);
INSERT INTO PERSON (FIRST_NAME, LAST_NAME, AGE,
COUNTRY)  VALUES ('John', 'Doe', 23, 'Italy');
INSERT INTO PERSON (FIRST_NAME, LAST_NAME, AGE,
COUNTRY)  VALUES ('Mary', 'Jane', 35, 'France');
```

```
INSERT INTO PERSON (FIRST_NAME, LAST_NAME, AGE,
COUNTRY)  VALUES ('Samuel', 'Felix', 28,
'Germany');

INSERT INTO PERSON (FIRST_NAME, LAST_NAME, AGE,
COUNTRY)  VALUES ('James', 'Smith', 51, 'United
States');
```

It creates a table called **`PERSON`** and inserts four rows into it. We will use this table to explore the **`Statement`** interface and its derivations, such as **`PreparedStatement`** and **`CallableStatement`**.

The **`Statement`** interface offers the most straightforward way of sending SQL queries to a relational database. By calling the method **`createStatement`** from the **`Connection`** class, we get a **`Statement`** object that lets us execute SQL queries in a connected database.

Consider the following example:

```
public class JdbcConnection {


        public static void main(String... args)
throws Exception {

            // Code omitted

            var statement =
connection.createStatement();

            printPerson(statement);

        }

        private static void printPerson(Statement
statement) throws

        SQLException {

            var retrieveAllPersons = "SELECT *
FROM PERSON";

            var result =
statement.executeQuery(retrieveAllPersons);

            while(result.next()) {
```

```
                var firstName =
result.getString("FIRST_NAME");
                var age = result.getLong("AGE");
                System.out.println("First Name:
"+firstName+"|Age: "+age);
            }
        }
        // Code omitted
}
```

The **printPerson** method receives a **Statement** object returned from the execution of the **createStatement** method from the **Connection** object. The **Statement** interface has the **executeQuery** method that receives a string representing the SQL query we want to execute in the database. In this example, we define a query to select all entries from the **PERSON** table. The query results are stored as a **ResultSet** object in the result variable.

In the previous example, we got the database results using the column's name:

```
var firstName = result.getString("FIRST_NAME");
var age = result.getLong("AGE");
```

It is also possible to retrieve data using the column's index:

```
var firstName = result.getString(0);
var age = result.getLong(2);
```

When using the index approach, we do not pass the column's name. Instead, we pass an index number that corresponds to the column position from which we want to retrieve the data.

Consider the following example to find a **PERSON** record that matches the **FIRST_NAME**:

```
private static void
findAndPrintPersonByFirstName(Statement statement,
String firstName) throws SQLException {
    var retrievePersonByFirstName = "SELECT * FROM
```

```
PERSON WHERE
    FIRST_NAME="+firstName;
    ResultSet result =
statement.executeQuery(retrievePersonByFirstName);
    while(result.next()) {
       String resultFirstName =
result.getString("FIRST_NAME");
       long resultAge = result.getLong("AGE");
       System.out.println("First Name:
"+resultFirstName+" | Age: "
       +resultAge);
    }
}
```

We use string concatenation to include the value from the **`firstName`** method parameter in the SQL statement. Everything is fine if we pass a legit name, as follows:

```
findAndPrintPersonByFirstName(connection.createStatement(),
"James");
// First Name: James | Age: 51
```

As expected, only the row containing James as the **`FIRST_NAME`** was returned. Things are okay if we have full control over the values we pass to a **`Statement`**. However, security problems may arise if we use data provided by the user to set the values of a **`Statement`**. It happens because the **`Statement`** approach is vulnerable to SQL injection attacks where an SQL code can be manipulated to produce unintended results, as follows:

```
findAndPrintPersonByFirstName(connection.createStatement(),
"James' OR '1'='1");
// First Name: John | Age: 23
// First Name: Mary | Age: 35
// First Name: Samuel | Age: 28
// First Name: James | Age: 51
```

In the example, we passed the string **`"James' OR '1'='1"`**. The **`Statement`** does not validate the values being passed, so the string part **`' OR '1'='1`** is interpreted as a component of the SQL statement, which is executed and returns all rows from the **`PERSON`** table.

Since the SQL statements produced by the **`Statement`** interface do not protect against SQL injection attacks, they are not recommended for scenarios where SQL statements are constructed using data from external sources. We can rely on the **`PreparedStatement`** interface for such scenarios.

Executing parameterized queries with the PreparedStatement

The **`PrepapredStatement`** interface extends from the **`Statement`** interface. SQL statements built using the **`PreparedStatement`** offer more flexibility and security. It is flexible because it lets us define SQL statements with parameterized values. It is secure because it can prevent SQL injection attacks by validating the value parameters. Consider the following example:

```
private static void
findAndPrintPersonByFirstName(String firstName)
throws SQLException {

    String sql = "SELECT * FROM PERSON WHERE
FIRST_NAME=?";


    PreparedStatement preparedStatement =
connection.prepareStatement(sql);

    preparedStatement.setString(1, firstName);


    ResultSet result =
preparedStatement.executeQuery();

        while (result.next()) {

        String resultFirstName =
result.getString("FIRST_NAME");

        long resultAge = result.getLong("AGE");
```

```
        System.out.println("First Name:
"+resultFirstName+" | Age: "
        +resultAge);
    }
}
```

The parameter is denoted by the `?` character in parts of the SQL statement where we want to use parameter values. We can set which value must be used for each parameter by calling methods like **`setString`** from the **`PreparedStatment`** interface. In our example, the **`setString`** receives the number **`1`**, which represents the index position of the `?` character, and a **`String`** object that is used as a value for the `?` character in the index position. The **`PreparedStament`** ensures protection against SQL injection attacks like the one we saw previously in the **`Statement`** approach. Consider the following call to the **`findAndPrintPersonByFirstName`** method:

```
findAndPrintPersonByFirstName("James' OR '1'='1"); //
Prints nothing
```

Attempts to use arbitrary SQL statements like **`' OR '1'='1`** are appropriately handled by the **`PreparedStament`** and not executed in the database. That is why it is always recommended to use **`PreparedStatement`** when constructing SQL statements using data coming from external sources.

Besides offering the **`Statement`** for simple queries and the **`PreparedStatement`** for parameterized queries, the JDBC API also has the **`CallableStament`** interface, which allows us to call stored procedures. Let us look at how it works.

Calling store procedures with the CallableStatement

The JDBC API offers the **`Statement`** for simple queries and the **`PreparedStatement`** for parameterized queries. It also has the **`CallableStament`** interface, which allows us to call stored procedures. Let us use the stored procedure described as follows to explore how the **`CallableStament`** works:

```
DELIMITER //

DROP PROCEDURE IF EXISTS findPersonOlderThanAge //
```

```
CREATE PROCEDURE findPersonOlderThanAge (IN AGE_PARAM
INT)

BEGIN

   SELECT FIRST_NAME, AGE FROM PERSON WHERE AGE >
AGE_PARAM;

END //

DELIMITER ;
```

This stored procedure takes the **AGE_PARAM** integer as the only parameter and returns all **PERSON** records where the **AGE** value column is greater than the **AGE_PARAM**. An example showing how to call that store procedure using the **CallableStatement** is as follows:

```
private static void
findAndPrintPersonOlderThanAge(int age) throws
SQLException {

   String sql = "CALL findPersonOlderThanAge(?)";

   CallableStatement callableStatement =
connection.prepareCall(sql);

   callableStatement.setInt(1, age);


   ResultSet result =
callableStatement.executeQuery();

   while (result.next()) {

      String resultFirstName =
result.getString("FIRST_NAME");

      long resultAge = result.getLong("AGE");

      System.out.println("First Name:
"+resultFirstName+" | Age: "

      +resultAge);

   }

}
```

The **CallableStatement** interface extends the **PreparedStatement** interface and supports statement

parametrization. In our example above, the **findPersonOlderThanAge** stored procedure expects an integer parameter representing age. We use the **CALL** SQL keyword before the stored procedure name. Note also that we are using the **?** character to set a parameter at **findPersonOlderThanAge(?).** Because **CallableStatement** extends the **PreparedStatement** interface, we can set the age parameter using the **setInt** method. We call the **findAndPrintPersonOlderThanAge** method, passing the **age** integer as a parameter:

```
findAndPrintPersonOlderThanAge(29);

// First Name: Mary | Age: 35

// First Name: James | Age: 51
```

As expected, the store procedure returned two records of persons older than 29.

You may have noted that in this section and the sections covering the **Statement** and **PreparedStatement** interfaces, we have been processing results using a **ResultSet** object. Let us examine this further.

Processing results with the ResultSet

The **ResultSet** is an interface that allows data to be retrieved from and persisted in the database. Whenever we execute the method **executeQuery** from a **Statement-type** object (which also includes **PreparedStatement** and **CallableStatement**), the result is a **ResultSet** object because the **executeQuery** method is used for **SELECT** statements whose purpose is to retrieve data from the database. The **executeUpdate** method is also used when making changes in the database with the **UPDATE** or **DELETE** statements. The **executeUpdate** returns an integer representing the number of rows affected by a **UPDATE** or **DELETE** statement. This section will focus on the **executeQuery** method and the **ResultSet** object it returns.

When querying data from a database, we are interested in the rows and columns returned by the query. The **ResultSet** interface allows us to inspect what a **SELECT** statement returned after it was executed in the database. The following command will help us get a **ResultSet**:

```
var retrieveAllPersons = "SELECT * FROM PERSON";

ResultSet resultSet =
statement.executeQuery(retrieveAllPersons);
```

The **ResultSet** stores the query result in a table-like structure that we can navigate through using a cursor that the **ResultSet** provides. This cursor starts right before the first row returned by the executed query. We can move the cursor position by using the **next** method from the **ResultSet**:

```
while (resultSet.next()) {

    String firstName =
resultSet.getString("FIRST_NAME");

    long age = resultSet.getLong("AGE");

    System.out.println("First Name: "firstName+" |
Age: "+age);

}
```

As the cursor starts right before the first row, when the while loop calls the **resultSet.next()** for the first time, it moves the cursor to the first row. While the **ResultSet** cursor is in the first table row position, we extract the data we want using the **resultSet.getString** and **resultSet.getLong** methods. These methods are mapped to the data type of the database columns. You should use the **getString** method to retrieve data stored in a **VARCHAR** column. If you store data in a **BOOLEAN** column, you should use the **getBoolean** to retrieve data from that column. Accessing a **ResultSet** column using the column index rather than the name is also possible. For example, instead of calling **resultSet.getString("FIRST_NAME")** we can call **resultSet.getString(1)**.

In our example, the **while** iteration continues until **resultSet.next()** returns false, meaning no more rows need to be processed. By default, the **ResultSet** cursor moves from beginning to end. And if something changes in the database while the **ResultSet** is being traversed, those changes will not be reflected in the **ResultSet** object. We can change this behavior by passing special properties when creating **Statement** objects. The following example shows how we can do that:

```
private static void printPerson() throws
SQLException {
    String sql = "SELECT * FROM PERSON";

    PreparedStatement preparedStatement =
connection.prepareStatement(
        sql,
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
        var resultSet =
preparedStatement.executeQuery();
    resultSet.last();
    resultSet.previous();

   String resultFirstName =
resultSet.getString("FIRST_NAME");
   long resultAge = resultSet.getLong("AGE");
   System.out.println("First Name:
"+resultFirstName+" | Age: "+resultAge);
}
...
printPerson() // First Name: Samuel | Age: 28
```

We can specify different **`ResultSet`** options when creating a **`Statement`** object. Here, we are passing the **`ResultSet.TYPE_SCROLL_SENSITIVE`** option to make the **`ResultSet`** scrollable—its cursor can move forward and backward. By default, **`ResultSet`** objects are created with the **`ResultSet.TYPE_SCROLL_INSENSITIVE`** option, which means that changes in the database are not reflected in the **`ResultSet`** object. In addition, we have the **`ResultSet.CONCUR_READ_ONLY`** option that makes this **`ResultSet`** read-only, meaning we cannot make database changes. To have a modifiable **`ResultSet`**, we need to pass the **`ResultSet.CONCUR_UPDATABLE`** option that allows us to

use the **`ResultSet`** to make database changes.

The following is an example of how we can update rows of data using the **`ResultSet`**:

```
private static void updatePersonCountry(String
firstName, String country) throws SQLException {
    String sql = "SELECT * FROM PERSON WHERE
FIRST_NAME="+firstName;


    PreparedStatement preparedStatement =
connection.prepareStatement(
        sql,
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    var resultSet =
preparedStatement.executeQuery();


    while (resultSet.next()) {
        String resultFirstName =
resultSet.getString("FIRST_NAME");
        String resultCountry =
resultSet.getString("COUNTRY");


        System.out.println("Before update -- First
Name: "
        +resultFirstName+" | Country:
"+resultCountry);
        resultSet.updateString("COUNTRY",
country);
        resultSet.updateRow();
        resultCountry =
resultSet.getString("COUNTRY");
```

```
        System.out.println("After update -- First
Name: "
        +resultFirstName+" | Country:
"+resultCountry);
    }
}
...
updatePersonCountry("James", "Canada");
// Before update -- First Name: James | Country:
United States
// After update -- First Name: James | Country:
Canada
```

Creating a statement with the **`ResultSet.CONCUR_UPDATABLE`** option is required to obtain a **`ResultSet`** object that can perform changes in the database. In the above example, where we want to update a person's country, we traverse the **`ResultSet`** that contains all results matching the previous **`SELECT`** statement that returns all rows where the first name is James. Once we are in the **`ResultSet`** cursor position of the row we want to change, we call first **`resultSet.updateString("COUNTRY", country)`** where **`"COUNTRY"`** is the column name and **`country`** is the variable containing the value we want to use for the update. Similar to **`get`** methods, the **`update`** methods from the **`ResultSet`** are also based on the database column types, providing methods like **`updateInt`**, **`updateBoolean`**, and so on. The update only persists in the database after we call **`resultSet.updateRow()`**.

The **`Statement`**, **`PreparedStatement`**, **`CallableStatement`**, and **`ResultSet`** interfaces are the building blocks of what the JDBC API provides to allow Java applications to handle relational databases. Relying purely on the JDBC API is for those cases where crafting SQL statements and directly manipulating the database results do not represent a maintenance burden.

When a Java application manages and executes many database statements, it may benefit from **object-relational mapping** (**ORM**) technologies that map database entities to Java objects, which can simplify the development and maintainability of the code

responsible for interacting with a database. A specification called **Jakarta Persistence API** (**JPA**) establishes how database entities should be managed in a Java application. Let us explore this in the next section.

Simplifying data handling with the Jakarta Persistence

Maintained by the *Eclipse Foundation*, the Jakarta Persistence (previously known as Java Persistence API/JPA) is a specification that provides a standard on how Java applications can handle database entities. It is part of the Jakarta EE, a project previously maintained by Oracle under the Java EE name. As a specification, the Jakarta Persistence only describes how a Java application should manage database entities. It provides the Jakarta Persistence API with a set of interfaces and classes that constitute the specification. Different providers implement the Jakarta Persistence specification. Hibernate and EclipseLink are well-known Jakarta Persistence implementations.

The decision to use Jakarta Persistence usually comes when a Java project requires a consistent database handling mechanism that allows database entities to be mapped to Java classes. The ability to represent a database table as a Java class enables developers to conveniently handle database entity relationships as Java objects without manipulating SQL code to achieve it because this is done by a Jakarta Persistence implementation like Hibernate, for example.

Another good reason to adopt Jakarta Persistence is the standardization benefits it brings. It allows applications to switch between database technologies without the need to refactor most of the code responsible for handling database entities. This happens because the Java representation of database entities is translated to work with different database technologies.

It means that you can, for most scenarios, change your Java application's database from Oracle to SQL Server, for example, and keep your code based on the Jakarta Persistence untouched.

To understand how we can tap into the benefits provided by Jakarta Persistence, we will learn how to implement Jakarta Persistence entities and their main characteristics. We will also learn how to

configure and use Hibernate, the most used ORM technology that implements the Jakarta Persistence specification.

Defining entities

The magic behind the Jakarta Persistence lies in mapping a Java class to a database table. We can do that by placing the Entity annotation on top of a Java class:

```
@Entity
@Table(name = "USER")
public class User {

    @Id
    @Column(name = "id", updatable = false,
nullable = false)
    private UUID id;

    @Column(name = "email", nullable = false)
    private String email;

    @Column(name = "password", nullable = false)
    private String password;

    @Column(name = "name", nullable = false)
    private String name;

    // Constructor, getters and setters omitted
}
```

The **`Entity`** annotation is mandatory to map a class to a database table. We can also use the **`Table`** annotation to specify the table name to which the Java class is mapped. If we do not specify the **`Table`** annotation, the Java class name is used in the mapping, and

that will work only if both the Java class and table have the same name. Otherwise, the mapping will not be possible because the table name differs from the Java class name. Nevertheless, using the `Table` annotation is recommended to ensure code clarity.

The class attributes are mapped to table columns. The first attribute we declare is the `id` having the `Id` annotation, which is mandatory for every entity class. Note that we have the `Column` annotation describing the column name and declaring that this column cannot be updated and does not support `NULL` values. As with the `Table` annotation, the `Column` annotation is optional, and if we do not specify it, a mapping will be attempted using the class attribute names against the actual table column names.

We can use the `User` entity class to retrieve and persist data of the `USER` table. This entity class gives us the possibility to handle database entities in the same way we handle other Java classes.

When dealing with relational databases, we usually work with multiple tables that relate to each other. Next, let us look at how to use Jakarta Persistence to define a relationship between entities.

Defining entity relationships

In relational databases, we arrange data in tables that may refer to each other. Table relationships can help us better structure the data of the problem domain we are dealing with. The Jakarta Persistence lets us represent table relationships using entity classes. We can do this through annotations denoting the possible entity relationships. Let us start our exploration by checking how the `OneToMany` relationship annotation works.

OneToMany

A one-to-many relationship means that a row in a table can be mapped to multiple rows in another table. To illustrate this, consider the scenario where we have the `AttributeDefinition` and `AttributeValue` entities used to map tables storing attribute definitions and their values. The `AttributeDefinition` stores the attribute type and name. The `AttributeValue` stores the attribute value and a reference to the attribute definition so we can know which attribute type and name the attribute value comes from.

The following is the SQL code for MySQL that we can use to create

tables representing attribute definitions and their values:

```
CREATE TABLE ATTRIBUTE_DEFINITION(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    type VARCHAR(255) NOT NULL
);
CREATE TABLE ATTRIBUTE_VALUE(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    definition_id INT NOT NULL,
    value VARCHAR(255) NOT NULL,
    FOREIGN KEY (definition_id) REFERENCES
ATTRIBUTE_DEFINITION(id)
);
```

The **`ATTRIBUTE_VALUE`** table has a foreign key **`definition_id`** referencing the **`id`** column from the **`ATTRIBUTE_DEFINITION`** table. Based on this table structure, the following is how we define the **`AttributeDefinition`** entity:

```
@Entity
@Table(name = "ATTRIBUTE_DEFINITION")
public class AttributeDefinition {

    @Id
    private Long id;
    private String name;
    private String type;
    @OneToMany(mappedBy="attributeDefinition")
    private List<AttributeValue> values;
    // Constructor, getters and setters omitted
}
```

We use the **`OneToMany`** annotation to express the relationship

where an **AttributeDefinition** refers to many **AttributeValue** entities; that is why the **values** attribute is of type **List<AttributeValue>**. The **mappedBy** property specifies the class attribute name used in the **AttributeValue** entity class to refer to the **AttributeDefinition** entity class. If the **AttributeDefinition** entity has a one-to-many relationship to the **AttributeValue**, it means the **AttributeValue** entity can have a many-to-one relationship to the **AttributeDefinition** entity. Let us see how it can be done next.

ManyToOne

A many-to-one relationship occurs when multiple rows of a table can be mapped to only one row of another. In the context of attribute definitions and values, we can have multiple attribute values of only one attribute type. We can implement the **AttributeValue** entity using the **ManyToOne** annotation as follows:

```
@Entity
@Table(name = "ATTRIBUTE_VALUE")
public class AttributeValue {

    @Id
    private Long id;
    private Long definition_id;
    private String value;
    @ManyToOne
    @JoinColumn(name="definition_id",
nullable=false)
    private AttributeDefinition
attributeDefinition;
    // Constructor, getters and setters omitted
}
```

Remember we specified the **attributeDefinition** in the **mappedBy** property of the **AttributeDefinition** entity class. The

attributeDefinition appears here as a class attribute of the **AttributeValue** entity class. Also note that in addition to the **ManyToOne** annotation, we have a **JoinColumn** annotation specifying which column from the **AttributeValue** entity, **definition_id** in our example, is used to map **AttributeValue** entities back to the **AttributeDefinition** entity.

We use the **ManyToOne** annotation here because this is a bidirectional relationship in which the **AttributeDefintion** entity owns the relationship with the **AttributeValue** entity.

Let us check next how to use Jakarta Persistence to implement a one-to-one relationship.

OneToOne

A one-to-one relationship occurs when a row from a table maps to only one row of another table.

Consider the scenario where we need to map the relationship between the **ACCOUNT** and **PROFILE** tables used in a database serving an internet forum. The SQL code for MySQL that creates the **ACCOUNT** and **PROFILE** tables is as follows:

```
CREATE TABLE PROFILE(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description VARCHAR(255) NOT NULL,
    website VARCHAR(255) NOT NULL
);
CREATE TABLE ACCOUNT(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    profile_id INT NOT NULL,
    FOREIGN KEY (profile_id) REFERENCES
PROFILE(id)
);
```

The **PROFILE** table must be created first because its **id** is a foreign key in the **ACCOUNT** table. Next, we can start by defining the **Account** entity class:

```
@Entity
@Table(name = "ACCOUNT")
public class Account {

    @Id
    @GeneratedValue (strategy =
GenerationType.IDENTITY)
    private Long id;
    private String email;
    private String password;
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "profile_id",
referencedColumnName = "id")
    private Profile profile;
   // Constructor, getters and setters omitted
}
```

We have the **GeneratedValue** annotation right below the **Id** annotation. It signals that the **ID** database column has an auto-increment mechanism for generating **ID** values. The **GenerationType.IDENTITY** strategy is used for scenarios where a special database identity column is used when a new entity is created and needs an **ID** value as the primary key. The Jakarta Persistence provider does not generate the **ID** value; instead, the underlying database generates the **ID**. There are other strategies like **GenerationType.AUTO**, **GenerationType.SEQUENCE**, and **GenerationType.TABLE** that provides different behaviors for ID generation.

When a new user registers in the internet forum, he gets an account and a profile. The account contains login data such as email and password, while the profile has data like name, description, and

website. Every account must have only one profile linked to it. We express it using the **OneToOne** annotation. Note the usage of the **cascade = CascadeType.ALL** property. Using it means that if an **Account** entity is deleted, then the deletion operation will be propagated to its child relationship entities, which implies that the **Profile** entity should also be deleted as it is a child entity from the **Account** parent entity.

In the **ACCOUNT** table, we have a **profile_id** column that acts as a foreign key that points to the **id** column in the **PROFILE** table. Such a relationship is represented using the **JoinColumn** annotation. Note that the **Account** entity class has a class attribute called **profile**. When defining the **Profile** entity class, we refer to the **profile** attribute:

```
@Entity
@Table(name = "PROFILE")
public class Profile {

    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    private String website;
    @OneToOne(mappedBy = "profile")
    private Account account;
    // Constructor, getters and setters omitted
}
```

Again, we use the **OneToOne** annotation to map the **Account** entity back to the **Profile** entity. The **mappedBy** contains the name of the class attribute **profile** defined in the **Account** entity class.

The last relationship to check is the many-to-many relationship. Let

us see how we can implement it using Jakarta Persistence.

ManyToMany

In many-to-many relationships, a row from one table can appear multiple times in another table and vice versa. Such a relationship usually occurs when a join table connects two tables.

Let us consider the scenario of a user management system that stores users, groups, and the user group membership. The following is the SQL code for MySQL that we can use to create tables to support the system:

```
CREATE TABLE USER(
    id UUID PRIMARY KEY NOT NULL,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE GROUP(
    id UUID PRIMARY KEY NOT NULL,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE MEMBERSHIP(
    id int NOT NULL AUTO_INCREMENT,
    user_id UUID NOT NULL,
    group_id UUID NOT NULL,
    PRIMARY KEY (user_id, group_id),
    FOREIGN KEY (user_id) REFERENCES USER(id),
    FOREIGN KEY (group_id) REFERENCES GROUP(id)
);
```

The **`MEMBERSHIP`** table is the join table we use to connect the **`USER`**

and **GROUP** tables. Note that the **MEMBERSHIP** table has a foreign key called **user_id** pointing to the **id** column from the **USER** table and another foreign key called **group_id** pointing to the **id** column from the **GROUP** table.

Let us start by implementing the **User** entity:

```
@Entity
@Table(name = "USER")
public class User {

    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(
            name = "UUID",
            strategy =
"org.hibernate.id.UUIDGenerator"
    )
    @ColumnDefault(" uuid()")
    private UUID id;
    private String email;
    private String password;
    private String name;
    @ManyToMany
    @JoinTable(
            name="MEMBERSHIP",
            joinColumns =
@JoinColumn(name="user_id"),
            inverseJoinColumns =
@JoinColumn(name="group_id")
    )
    private List<Group> groups;
```

```
    // Constructor, getters and setters omitted
}
```

Instead of using a number as the ID, we use a UUID. Since UUIDS are not numbers, we cannot rely on ID generators like **`GenerationType.IDENTITY`** or **`GenerationType.AUTO`**. That is why we have the **`GenericGenerator`** annotation using the **`org.hibernate.id.UUIDGenerator`** class, provided by Hibernate (which we will explore further in the next section) and not Jakarta Persistence, as the generator strategy. Note the **`ColumnDefault`** annotation; it defines a default column value when none is provided. We pass the **`uuid()`** function from the MySQL database that generates random UUID values. Such a function can appear under different names depending on your database technology.

We have a group class attribute annotated with the **`ManyToMany`** annotation, followed by a **`JoinTable`** annotation that specifies the join table name as **`MEMBERSHIP`**. The **`joinColumns`** property refers to the **`user_id`** column, and the **`inverseJoinColumns`** refers to the **`group_id`** column, both columns from the **`MEMBERSHIP`** table.

Next, we define the **`Group`** entity:

```
@Entity
@Table(name = "GROUP")
public class Group {
    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(
            name = "UUID",
            strategy =
"org.hibernate.id.UUIDGenerator"
    )
    @ColumnDefault("uuid()")
    private UUID id;
    private String name;
    @ManyToMany(mappedBy = "groups")
```

```
    private List<User> users;
    // Constructor, getters and setters omitted
}
```

The **`ManyToMany`** is used again, but the **`Group`** entity is mapped back to the **`User`** entity through the **`mappedBy`** property.

Now that we have covered the fundamentals of defining entities and their relationships, let us explore how to use *Hibernate* to retrieve and persist database entities.

Using Hibernate to handle database entities

As stated earlier, the Jakarta Persistence is only a specification; it does not provide an implementation we can use to interact with databases, so we need to rely on one of the available Jakarta Persistence implementations. Hibernate is the most well-known Jakarta Persistence implementation, allowing Java applications to connect and handle database entities using the Jakarta Persistence specification.

To get started, let us see how we can configure Hibernate.

If your Java application is configured as a Maven project, you can add the following dependency to the **`pom.xml`** of your project:

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.4.Final</version>
</dependency>
```

You can also download the Hibernate ORM library[2] and put it in your Java application's class or module path.

The persistence layer between the Java application and the database is defined by the **`persistence.xml`** file located in the **`resources/META-INF`** directory of a Java project, as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/
persistence"
```

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/
persistence
 http://java.sun.com/xml/ns/persistence/
persistence_1_0.xsd"
 version="1.0">

   <persistence-unit name="user">

<provider>org.hibernate.jpa.HibernatePersistenceProvider
provider>
     <properties>
       <property
name="jakarta.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/user"/>
       <property
name="jakarta.persistence.jdbc.user" value="test"/
>
       <property
name="jakarta.persistence.jdbc.password"
value="test"/>
     </properties>
   </persistence-unit>


</persistence>
```

This file specifies a persistent unit called the **`user`**. Inside it, we can define which Jakarta Persistence provider we want to use: the **`org.hibernate.jpa.HibernatePersistenceProvider`** in our case. It is in the **`persistence.xml`** that we also define the database connection details. The configuration provided by the **`persistence.xml`** file enables the creation of the **`EntityManager`**, an object responsible for initiating the interaction with a database:

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("user");

var entityManager =
entityManagerFactory.createEntityManager();
```

When we call `Persistence.createEntityManagerFactory("user")`, Hibernate tries to identify in the `persistence.xml` file a persistence unit called `user`. From an `EntityManagerFactory` object, we call `createEntityManager` to get an `EntityManager` that can be used for further interactions with the database. The `EntityManager` allows sending queries to the database:

```
List<Account> account = entityManager

    .createQuery("SELECT a FROM Account a",
Account.class)

    .getResultList();
```

The SQL query we pass to the `createQuery` method uses the name of the `Account` entity class rather than the `ACCOUNT` table name. Also, we pass the `Account.class` to ensure that Hibernate returns `Account` entities. The `getResultList` is used when the query returns one or more results stored in a `List` collection.

It is also possible to create SQL queries that refer to real table names using the `createNativeQuery`:

```
List<Account> account = entityManager

    .createNativeQuery("SELECT * FROM ACCOUNT",
Account.class)

    .getResultList();
```

We can update or create rows in the database using the persist method:

```
private void persist(Account account) {

    entityManager.persist(account);

}
```

The `persist` method expects, as a parameter, an entity class that is mapped to a database table.

Next, we have the `remove` method, which can be used to delete

database rows using the following command:

```
private void persist(Account account) {
    entityManager.remove(account);
}
```

The **`remove`** method uses the entity's primary key to identify and remove the table row.

In addition to the **`EntityManager`**, which comes from the Jakarta Persistence specification, we can also use the **`Session`**, which is a Hibernate-specific interface that extends the **`EntityManager`** interface:

```
Session session =
entityManager.unwrap(Session.class);
List<Account> account = session
    .createNativeQuery("SELECT * FROM ACCOUNT",
Account.class)
    .getResultList();
```

A **`Session`** object is recommended when the application relies on special features available only in Hibernate. Otherwise, the **`EntityManager`** is enough.

Let us explore next the **Jakarta Persistence Query Language** (**JPQL**) and the Criteria API.

Exploring JPQL and the Criteria API

Besides providing the **object-relationship mapping** (**ORM**) capabilities we have been exploring, the Jakarta Persistence also offers the JPQL. Instead of constructing SQL statements that refer to real database tables, we can use JPQL to handle Java objects that map to real database tables.

Consider the following example where we use JPQL:

```
public User findByEmail(String email) {
    Query query = session.createQuery("SELECT u
FROM User u WHERE email
    = :email", User.class);
```

```
    query.setParameter("email", email);
    return (User) query.getSingleResult();
 }
```

The JPQL query refers to the **`User`** Java entity instead of the database table name. The **`User`** is assigned to the **`u`** identifier, also known as an alias, of the entity we are dealing with. The **`:email`** is defined as a JPQL placeholder; we provide a value for such a placeholder using the **`setParameter`** from the **`Query`** type.

The Criteria API is an alternative to JPQL that lets us define query constraints. Below is an example showing how we can rewrite the previous JPQL-based example using the Query API:

```
public User findByEmail(String email) {
    CriteriaBuilder criteriaBuilder =
session.getCriteriaBuilder();
    CriteriaQuery<User> criteriaQuery =
criteriaBuilder.createQuery
    (User.class);
    Root<User> user =
criteriaQuery.from(User.class);

    criteriaQuery
    .select(user)

.where(criteriaBuilder.equal(user.get("email"),
email));

    return
session.createQuery(criteriaQuery).getSingleResult();
}
```

We create a **`CriteriaBuilder`** that builds a **`CriteriaQuery`** object parameterized to work with **`User`** entities. From the **`CriteriaQuery`**, we get the **`Root`** object for the **`User`** entity. With **`CriteriaQuery`**, we express the database interaction using the

select and where methods. The constraint to select `User` entities having an specified email address is applied inside the where method using the equal method from the `CriteriaBuilder`. Besides `equal`, the `CriteriaBuilder` provides other interesting helper methods like `notEqual`, `isNull`, and `isNotNull`, to name a few.

In this section, we saw how to create Jakarta Persistence entities and their relationships. We learned how to configure and use Hibernate, the most well-known Jakarta Persistence implementation, to handle entity classes by querying, persisting, and removing them. We also explored JPQL and the Criteria API.

This section covered the fundamentals required to prepare a Java application to take advantage of the Jakarta Persistence specification and ORM technologies like Hibernate.

The following section will cover the approaches to providing databases while developing an application locally.

Exploring local development approaches when using databases

When developing an application that depends on a database, we need to define how the database will be provided so the application can properly start and connect to it. In this section, we will assess three different approaches: remote databases, in-memory databases, and container databases.

Local development with a remote databases

When developing an application that depends on a database, we can rely on remote databases for local development. The problem with this approach is the operational burden of creating and managing those databases. Also, we need to consider the latency issues that may occur because the local application will access the database through the network.

Remote databases are employed when providing the database locally is not feasible. It can happen, for example, because the database server is too big to run on the developer's laptop.

Local development with in-memory databases

Using in-memory databases is a lightweight approach to locally run and test applications that depend on a database. In-memory databases are temporary and last for as long as the application runs. Once the application is terminated, the in-memory database is destroyed. Most in-memory database technologies, like the H2, offer the possibility to store the database in a file, so the data is preserved in a file instead of being destroyed when the application terminates.

The problem with the in-memory database approach is that some features in the actual database may not exist in the in-memory database, which can be a serious risk because the code used in the in-memory database may not work with the actual database and vice versa. So, in-memory databases are recommended for prototyping or testing basic database functionalities.

Local development with container databases

That is the preferable approach because it allows simple provisioning of different database servers using container technologies like Docker. Local applications can immediately consume databases bootstrapped with Docker.

The choice between these three approaches will vary according to constraints like computing resources and the database technology used by the application.

Compiling and running the sample project

As a sample project, we have a Java application called User Management. Backed by an in-memory database, this application shows how we can use Jakarta Persistence and Hibernate to create a system to manage user and group data from a relational database.

The project's source code is available at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2003**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

To compile the project, go to the *Chapter 3* directory from the book's repository. From there, you need to execute the following command:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the following command:

```
$ java -jar target/chapter03-1.0-SNAPSHOT-jar-with-
dependencies.jar
```

```
[User{email='admin@davivieira.dev', name='Admin',
groups=[Group{name='Administrators'}]},
User{email='user1@davivieira.dev', name='User 1',
groups=[Group{name='Administrators'},
Group{name='Users'}]},
User{email='user2@davivieira.dev', name='User 2',
groups=[Group{name='Users'}]},
User{email='user3@davivieira.dev', name='User 3',
groups=[Group{name='Users'}]}]
```

Conclusion

In this chapter, we learned how **Java Database Connectivity** (**JDBC**) provides the building blocks for database interactions. Seeking a consistent and standardized way to handle databases, we also learned about the Jakarta Persistence specification and one of its main implementations, Hibernate. We finished this chapter by assessing the possible approaches to accessing a database while locally running a Java application.

In the next chapter, we will cover the technologies and techniques used to test Java code. We will look at essential topics like unit and integration tests, the prominent testing framework JUnit 5, and the ability to test using real systems with Testcontainers.

---

1 **https://github.com/h-thurow/Simple-JNDI**

2 (check **https://hibernate.org/**)

Chapter 4

Preventing Unexpected Behaviors with Tests

Introduction

A fundamental success aspect of any Java project lies in how developers handle tests. They can handle tests using different approaches like unit and integration testing. The time invested in testing pays off by the number of prevented bugs. So, in this chapter, we will explore how to use technologies like JUnit 5 and Testcontainers to implement helpful unit and integration tests that prevent unexpected system behaviors, help us better understand application behaviors, and design simple yet effective code. While exploring testing technologies, we will also learn about good practices for writing helpful tests that are easy to grasp and maintain.

Structure

The chapter covers the following topics:

- Overviewing unit and integration tests
- Using JUnit 5 to write effective unit tests
- Implementing reliable integration tests with Testcontainers
- Compiling and running the sample project

Objectives

By the end of this chapter, you will have the fundamental skills to write effective unit and integration tests on Java applications using JUnit 5 and Testscontainers. Such skills will make you a better developer, ready to tackle any programming challenges with much more confidence by ensuring the features you are developing are secured by automated tests. Let us embark together on this fascinating testing journey.

Overviewing unit and integration tests

Common steps involved in developing a new application or adding features to an existing one consist of finding some way to run the application locally to observe how it behaves. Having the application running locally allows us to test application behaviors manually. After introducing some changes in the code, we recompile it, start the application, and perform some checks to confirm the changes are working as expected. However, this manual test approach depends on a human's ability to interpret the system behavior and judge whether it produces correct or wrong results. Besides being time-consuming, manual tests may not be sustainable in the long run. As the application receives more features, more manual testing will be required.

The idea is to find ways to automate some of the tests we would execute manually to ensure the application is working well. To achieve test automation, we can rely on techniques like unit and integration testing, which have different purposes in terms of testing scope, but both share the intent to automate tests. So, in this session, we will examine what it means to employ unit or integration tests in a Java application. Let us start by exploring unit tests.

Unit tests

When discussing tests, it is always important to consider the scope we want to address when validating system features. Such consideration is essential because the bigger the scope, the broader the test dependencies, which may include different systems, including databases, front-end applications, APIs, and so on. Another thing to consider when widening the scope is the cost and complexity associated with testing broader aspects of a system.

With the awareness that tests encompassing a large scope of dependencies are complex and costly, we can reflect on which system elements can be tested in a smaller scope. This leads us to the behaviors an application may expose through methods containing a sequence of instructions. At the core of any Java application, we have a collection of classes and methods orchestrating a sequence of instructions to produce useful application behaviors aimed at hopefully solving real-life problems.

With that in mind, unit tests are a technique to validate application behaviors on an isolated, self-contained level. At this level, we usually have methods containing logic that dictates how accurate and well an application handles its use cases. Unit tests are not concerned with the application's behavior when interacting with external resources like a database. Instead, unit tests validate application behaviors that can be checked without needing external resources or dependencies. That is why, for example, we can employ the so-called mocks to simulate such external resources when unit testing an application functionality that depends on external resources.

We can then state that unit tests are not supposed to test an entire use case or application feature but rather a more minor part that contributes to such a use case or feature. It is also important to understand that unit tests cannot replace other tests, like acceptance tests, which validate how an application behaves from the user's perspective.

Unit tests are so appealing because they are cheap to implement yet so beneficial. When properly implemented, they can protect critical areas of the application from unwanted changes and side effects. As the code complexity increases, unit tests can also help us identify what the application is doing with test cases targeting all the

required behaviors to deliver helpful application features.

Although unit tests help validate how units of code behave, they cannot help us answer how the application behaves when it needs to deal with external resources like a database or an API. For such a purpose, we can rely on integration tests, which we will examine next.

Integration tests

Earlier in this chapter, we discussed this manual test idea, which consists of locally running an application and observing how it behaves when something is changed. We can also locally run an application and see how it interacts with its external dependencies, like databases or APIs served by other systems. This validation increases our confidence that the application correctly handles data provided by external resources.

Besides checking how an application behaves when dealing with external dependencies, we can manually validate how different application components or modules interact. For example, imagine an application based on the API, service, and persistence layers materialized as application modules. We can validate how a request arrives at the API layer, passes through the service and persistence layers, and returns with some response data. Again, we can locally run our application and manually test this flow by preparing testing data, requesting payloads, and ensuring all external dependencies are in place to ensure our tests will not fail.

Real value can be achieved if we can automate the validation of application behaviors that span different modules and external dependencies. We are no longer interested only in the unit level of how an application behaves; instead, we want to validate the end-to-end behavior across application modules and their dependencies. We use integration tests to automate the validation of end-to-end system functionalities, considering their required dependencies.

Note that the integration test scope is broader than that of a unit test, which focuses on self-contained units of code responsible for supporting an application use case. With integration tests, we can validate how different units of code work together when combined.

In the next sections of this chapter, we will explore the techniques

and technologies we can use to create unit and integration tests. Let us start by exploring what JUnit 5 can offer in terms of unit test automation.

Using JUnit 5 to write effective unit tests

First introduced in 1997, JUnit remains the most widely used unit testing framework for Java applications. The latest JUnit 5 release brings enhancements and significant changes over its previous JUnit 4 release, but the fundamental unit testing principles remain the same. Our focus in this section is on JUnit 5 because of its up-to-date testing features and because modern Java applications can only benefit from the fantastic testing capabilities offered by JUnit 5. Let us start by learning to set up JUnit 5 in a Java project.

Setting up JUnit 5

The JUnit 5 framework is built into a modular structure, allowing Java projects to rely only on the framework modules that are relevant to the project. We can select those modules when defining the Java project's dependencies through a dependency manager like Maven or Gradle. The following code is an example showing which dependencies we must include in the Maven's pom.xml file of the Java project to have JUnit 5 working correctly:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.10.2</version>
     <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.10.2</version>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-launcher</
artifactId>
    <scope>test</scope>
    <version>1.8.2</version>
</dependency>
```

The dependencies with **`artifactId junit-jupiter-api`** and **`junit-jupiter-api`** are mandatory and represent the minimum requirement to get started with JUnit 5. For example, the last dependency with **`artifactId junit-platform-launcher`** can be included if you face issues while executing tests directly through an IDE like IntelliJ.

Aside from including the JUnit5 dependencies in the **`pom.xml`** file, we also need to add the **`maven-surefire-plugin`** to the plugin configuration of the **`pom.xml`**:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
     <version>3.2.5</version>
</plugin>
```

The **`maven-surefire-plugin`** allows us to execute tests when using Maven to build a Java application.

With all dependencies in place, we are ready to start creating unit tests. Before we do so, let us create a simple account registration application that will serve as the basis for our unit test exploration.

Introducing the account registration system

To allow further exploration of what we can do using JUnit 5, let us first create a simple account registration application that we will test later on. The registration system receives new account request payloads containing email, password, and birth date. It then validates the payload, saves it into a database, and returns the account data as a response, including the creation timestamp and

the account status. Start by defining the **`AccountPayload`** record class:

```
public record AccountPayload(
    String email,
    String password,
    LocalDate birthDate) { }
```

The **`AccountPayload`** represents the data provided by a client attempting to create a new account. Following it, we have the **`ValidatorService`** class:

```
public class ValidatorService {
    public final static String
INVALID_EMAIL_MESSAGE = "Email format is not
    valid";
    public final static String
INVALID_PASSWORD_MESSAGE = "Password must
    have at least 6 characters";
    public final static String
INVALID_BIRTHDATE_MESSAGE = "Age must be at
    least 18 years old";

    public void validateAccount(AccountPayload
accountPayload) throws
    Exception {
        var isEmailValid =
validateEmail(accountPayload.email());
        if(!isEmailValid) {
            throw new
Exception(INVALID_EMAIL_MESSAGE);
        }
        var isPasswordValid =
validatePassword(accountPayload.password());
```

```
        if(!isPasswordValid) {
            throw new
Exception(INVALID_PASSWORD_MESSAGE);
        }

        var isBirthDateValid =

validateBirthDate(accountPayload.birthDate());
        if(!isBirthDateValid) {
            throw new
Exception(INVALID_BIRTHDATE_MESSAGE);
        }
    }
    // Code omitted
}
```

The **INVALID_EMAIL_MESSAGE**, **INVALID_PASSWORD_MESSAGE**, and **INVALID_BIRTHDATE_MESSAGE** are constants we use to store messages returned when the validation fails. After the constants, the **validateAccount** method receives an **AccountPayload** object as a parameter. This method relies on the **validateEmail**, **validatePassword**, and **validateBirthDate** to check if the **AccountPayload** data is valid. If the data is invalid, it throws exceptions with a message describing why the validation could not be performed. We describe the implementation of the validation methods, as follows:

```
public class ValidatorService {
    //** Code omitted **//
    private boolean validateEmail(String email) {
        var regexPattern = "^(.+)@(\\S+)$";
        return Pattern.compile(regexPattern)
                .matcher(email)
                .matches();
```

```
    }
    private boolean validatePassword(String
password) {
        return password.length() >= 6;
    }
    private boolean validateBirthDate(LocalDate
birthDate) {
        return Period.between(birthDate,
LocalDate.now()).getYears() >= 18;
    }
}
```

The **`validateEmail`** checks if the email provided is in the correct format. The **`validatePassword`** ensures the password contains at least six characters. Finally, the **`validateBirthDate`** ensures that only birth dates equal to or above eighteen years old are accepted.

These two classes are enough to implement the account registration system initially. Let us start testing it then.

Testing the account registration system

To test the **`ValidatorService`** class, we create the **`ValidatorServiceTest`** test class in the **`dev.davivieira.account.service`** package from the Java project's **`src/main/test/java`** directory, as follows:

```
public class ValidatorServiceTest {


    private final ValidatorService
validatorService = new
    ValidatorService();
    // Code omitted
}
```

We start by setting and initializing the **`validatorService`** instance attribute with an instance of the **`ValidatorService`** that we will use to execute our unit tests.

Let us implement the first test responsible for testing if email validation is working by using the following code:

```
public class ValidatorServiceTest {

    private final ValidatorService
validatorService = new

    ValidatorService();


    @Test

    public void
givenInValidEmailString_thenValidationThrowsException()
{

        // Arrange

        var email = "@daviveira.dev"; // Invalid
email address

        var password = "123456";

        var birthDate = LocalDate.of(1980, 1, 1);


        // Prepare

        var accountPayload =
getAccountPayload(email, password, birthDate);


        // Execute and Pre-assert

        Exception exception =
assertThrows(Exception.class, () -> {


validatorService.validateAccount(accountPayload);

        });

        // Post-assert

        String expected = "Email format is not
valid";

        String actual = exception.getMessage();
```

```
            assertEquals(actual, expected);
        }
        // Code omitted
        }
}
```

We place the **`org.junit.jupiter.api.Test`** annotation on top of the methods we want to test. The test above checks if an exception is thrown when we pass **`AccountPayload`** with an invalid email address. Also, it checks if the exception message is indeed the one that says the email format is incorrect. Note that we use comment terms like *Arrange*, *Prepare*, *Execute*, *Pre-assert*, and *Post-assert* to describe the different stages of the testing method. This approach derives from the **Arrange-Act-Assert** (**AAA**) testing pattern that prescribes a way to organize tests to make them easier to read and maintain. Following, we explore the AAA pattern further.

The Arrange-Act-Assert pattern

The original AAA pattern is based on the following steps:

- The Arrange step arranges the data dependencies required to execute the test.
- On the Act (or Execute) step, we trigger the application behavior we want to test by usually calling an object's method. That call usually returns a result that we use on the Assert step.
- Based on the result returned from the Act step, in the Assert step, we check if the application behaved as we expected.

It is common to find variances in the AAA pattern consisting of additional steps to clarify even more what the test is doing. Our test contains the additional Preparation, Pre-assert, and Post-assert steps. There is no general rule of thumb for organizing your tests but expressing it through steps can benefit everyone involved in maintaining them.

The crucial part of any test method is performing assertions. Next, we will check how JUnit 5 helps us assert testing data.

Assertions

As one of the cornerstones of the JUnit 5 framework, the **`assertX`** methods validate the test behaviors and data. In the **`givenInValidEmailString_thenValidationThrowsException`** test, we are using the **`assertThrows`** to check if the **`validatorService.validateAccount(accountPayload)`** throws an **`Exception`** when we pass an **`AccountPayload`** with an invalid email:

```
// Execute and Pre-assert

Exception exception =
assertThrows(Exception.class, () -> {


validatorService.validateAccount(accountPayload);

});
```

Moving forward, there is also the **`assertEquals`**, which receives two parameters, the first parameter **`actual`** representing the actual value returned as the result of testing the method we are interested in, and the second parameter **`expected`** representing the value we expect the actual value to be:

```
// Post-assert

String expected = "Email format is not valid";

String actual = exception.getMessage();

assertEquals(actual, expected);
```

If the **`actual`** and **`expected`** variables match, then the assertion is successful; otherwise, the test fails. JUnit 5 provides plenty of other **`assertX`** methods, such as **`assertTrue`**, **`assertFalse`**, and others, allowing you to perform many different assertions.

There are situations when we need to unit test a method that contains calls to external resources like a database. We can overcome it by mocking those calls and allowing the test to execute without issues. Let us explore mocking techniques using a technology called Mockito.

When to use Mockito

The best unit testing scenario is when we test methods that do not

depend on external dependencies like databases or APIs. In such scenarios, the method we want to test contains a self-contained sequence of instructions we can directly call from our test scenario without being concerned that the execution may fail because some dependencies are unavailable during test runtime. When that is not the case, and we want to test a method that contains one or more calls to external dependencies, then we can mock those calls, allowing the application to continue with its execution and test the portion of the code we consider relevant.

Mockito can be integrated with JUnit 5. Next, we check how to set up this integration.

Setting up Mockito with JUnit 5

To get started with Mockito, we need to add the following dependencies to the **`pom.xml`** file:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

The dependency specified by the **`artifactId mockito-core`** provides the Mockito engine to the Java project, and the other dependency, **`mockito-junit-jupiter`**, enables us to use Mockito with JUnit 5.

Before exploring Mockito further, let us extend the account registration system by adding an external call that we will mock

later.

Adding an external call to the account registration system

A common external call is the one that persists data in the database. On the account registration system, such a call is triggered from a repository class:

```
public class AccountRepository {


    public void persist(Account account) {
        throw new RuntimeException("Database
integration is not implemented
        yet");
    }
}
```

Since we do not yet have a database integrated with the application, we throw a **`RuntimeException`** when the **`persist`** method is called.

Next, we implement the **`RegistrationService`**, responsible for validating the **`AccountPayload`**, creating an **`Account`** object, persisting it into the database, and returning the **`Account`** object. We start the implementation by defining the class attributes and constructor as follows:

```
public class RegistrationService {


    private final ValidatorService
validatorService;
    private final AccountRepository
accountRepository;


    public RegistrationService(ValidatorService
validatorService,
    AccountRepository accountRepository) {
        this.validatorService = validatorService;
```

```
        this.accountRepository =
accountRepository;
    }
    // Code omitted
}
```

We have the **`ValidatorService`** we implemented previously, and the **`AccountRepository`** intends to persist data in a database. Let us now implement the logic that registers the account:

```
public class RegistrationService {
    //Code omitted
    public Account register(AccountPayload
accountPayload) throws Exception
    {

validatorService.validateAccount(accountPayload);
        return createAccount(accountPayload);
    }


    private Account createAccount(AccountPayload
accountPayload) {
        var account = new Account(
                accountPayload.email(),
                accountPayload.password(),
                accountPayload.birthDate(),
                Instant.now(),
                Status.ACTIVE
        );
        accountRepository.persist(account); //
Throws a RuntimeException
        return account;
    }
```

```
}
```

The register method receives the **`AccountPayload`** object as a parameter that is validated on **`validatorService.validateAccount(accountPayload)`**. If the validation is okay, then it creates an **`Account`** object and persists it into the database by calling **`accountRepository.persist(account)`**. Finally, it returns the **`Account`** object. Calling the register method in a unit test will make it fail because of the **`RuntimeException`**. Let us see how to address it using Mockito.

Mocking external calls with Mockito

The **`RegistrationService`** class has a method called register that is responsible for the actual registration of new accounts. Let us then create the **`RegistrationServiceTest`** class to test the execution of the register method. We start by defining instance variables and initializing the mocks:

```
@ExtendWith(MockitoExtension.class)
public class RegistrationServiceTest {

    private RegistrationService
registrationService;

    @BeforeEach
    private void init(@Mock AccountRepository
accountRepository) {
        registrationService = new
RegistrationService(new
        ValidatorService(), accountRepository);

doNothing().when(accountRepository).persist(any());
    }
    // Code omitted
}
```

We use the class-level annotation **`@ExtendWith(MockitoExtension.class)`** to make Mockito capabilities available while executing JUnit 5 tests. Next, we add the **`@BeforeEach`** annotation above the **`init`** method. The idea behind this annotation is that the **`init`** method will be executed before every testing method in the class is executed. Other variations like **`@BeforeAll`**, **`@AfterEach`**, and **`@AfterAll`** allow us to execute helpful logic across different stages of the test life cycle.

Note that the **`init`** method receives the **`@Mock AccountRepository acountRepository`** as a parameter. The **`@Mock`** annotation marks the **`accountRepository`** parameter as a mock, which Mockito injects during test execution. Inside the **`init`** method, we create a new instance of the **`RegistrationService`**, passing a real instance of the **`ValidatorService`** and a mock of the **`AccountRepository`**.

Creating mocks is not enough; we need to specify in which conditions the mock will be used. Mockito enables us to define the application's behavior when a method from the mocked object is called. Remember that the persist method from the **`AccountRepository`** throws an exception:

```
public void persist(Account account) {
    throw new RuntimeException("Database
integration is not implemented
    yet");
}
```

To avoid this exception, we instruct Mockito to do nothing when the **`persist(Account account)`** method is called:

```
doNothing().when(accountRepository).persist(any());
```

The idea behind the construct above is that we state the desired behavior, **`doNothing()`**, followed by the condition **`when(accountRepository).persist(any())`**, where we pass the mocked object **`accountRepository`** and describe which method from it we are mocking, which is the **`persist(any())`** method. The **`any()`** is an argument matcher that instructs Mockito to accept any object passed as a parameter to a given method.

Having prepared the mocks, we can proceed to create our test using the following ccode:

```
@ExtendWith(MockitoExtension.class)
public class RegistrationServiceTest {
    // Code omitted
    @Test
    public void
givenValidAccountPayload_thenAccountObjectIsCreated()
    throws Exception {
        // Arrange
        var email = "user@daviveira.dev";
        var password = "123456";
        var birthDate = LocalDate.of(1980, 1, 1);

        // Prepare
        var accountPayload =
getAccountPayload(email, password, birthDate);
        // Execute
        var account =
registrationService.register(accountPayload);
        // Assert
        assertAll("Account is properly created",
                () ->
assertEquals(account.email(), email),
                () ->
assertEquals(account.password(), password),
                () ->
assertEquals(account.birthDate(), birthDate),
                () ->
assertEquals(account.status(), ACTIVE)
        );
```

```
    }
    // Code omitted
}
```

The test above aims to check if the **`Account`** object returned by calling **`registrationService.register(accountPayload)`** contains all the expected data. Note we have multiple **`assertEquals`** calls inside the **`assertAll`** method. Even if one of the **`assertEquals`** fails, the **`assertAll`** will continue executing the other assertions until the end and report which assertions have failed.

Next, let us check how to use Maven to execute the tests we created with the **`RegistrationServiceTest`** and **`ValidatorServiceTest`** testing classes.

Executing tests with Maven

We can execute tests directly through our preferred IDE or build tools like Maven or Gradle. Remember, to run JUnit 5 tests with Maven, the **`maven-surefire-plugin`** plugin must be configured in the **`pom.xml`** file.

We can use the following command to execute our tests with Maven:

```
$ mvn test
```

The above command will produce an output similar to the following one:

```
[INFO]
-------------------------------------------------------

[INFO]  T E S T S

[INFO]
-------------------------------------------------------

[INFO] Running
dev.davivieira.account.service.ValidatorServiceTest

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 0.028 s -- in
dev.davivieira.account.service.ValidatorServiceTest

[INFO] Running
```

```
dev.davivieira.account.service.RegistrationServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 0.480 s -- in
dev.davivieira.account.service.RegistrationServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped:
0
[INFO]
[INFO]
------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------
[INFO] Total time:  1.191 s
[INFO] Finished at: 2024-03-24T19:06:45+01:00
[INFO]
------------------------------------------------------------
```

JUnit5 and Mockito offer us many ways to unit test our applications. In this session, we have covered the essential features that help us leverage the benefits of unit testing. In the next section, we will learn how to use Testcontainers to create integration tests.

Implementing reliable integration tests with Testcontainers

The decision to use integration tests comes when we want to validate the interaction between different application components and how the application interacts with external dependencies like a database, for example. Integration tests are more expensive than unit tests because they require practically all the same dependencies the application would need if deployed somewhere. The increased setup cost comes with the benefit of running tests against real resources and data instead of using mocks when a resource is not available.

With the popularization of container technologies like Docker, which enables us to bring up any system in seconds, Java projects have started to leverage containers to simplify the implementation of integration tests. The Testcontainers library stands out as one of the most well-known libraries providing containers for testing purposes. It provides a state-of-the-art solution that lets developers easily integrate containers into their integration tests.

In this section, we will learn how to configure Testcontainers in our account registration application and use them to provide a MySQL database container used by an end-to-end integration test.

Setting up Testcontainers

Like JUnit 5, Testcontainer is built into a modular structure so that we can bring only the relevant dependencies to our Java project. Let us start by adding the Testcontainer dependencies in the Maven's **`pom.xml file`**, as follows:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mysql</artifactId>
    <version>1.19.7</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.19.7</version>
    <scope>test</scope>
</dependency>
```

The first dependency identified by the **`artifactId mysql`** allows the creation of MySQL database containers. The next dependency, **`junit-jupiter`**, enables us to add containers into the JUnit 5 tests life-cycle.

Before implementing integration tests, let us provide a Jakarta

Persistence configuration to ensure the account registration system can connect to a real MySQL database.

Integrating the account registration system with MySQL

We intend to connect the account registration system to a MySQL database. To do it, we need the following dependencies in the **`pom.xml`** file:

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.4.Final</version>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.3.0</version>
</dependency>
```

The first dependency, **`hibernate-core`**, provides the Jarkarta Persistence implementation we rely on to interact with a MySQL database. The second dependency, **`mysql-connector-j`**, provides the JDBC driver Jakarta Persistence requires. Next, we configure the **`persistence.xml`** file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/
persistence"
             xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
             xsi:schemaLocation="http://
java.sun.com/xml/ns/persistence http://
java.sun.com/xml/ns/persistence/
persistence_2_0.xsd"
             version="2.0">
```

```
    <persistence-unit name="account" transaction-
type="RESOURCE_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider
provider>
        <properties>
            <property
name="jakarta.persistence.jdbc.driver"

value="com.mysql.cj.jdbc.Driver" />
            <property
name="jakarta.persistence.jdbc.url"
                        value="jdbc:mysql://
localhost:3306/account" />
            <property
name="jakarta.persistence.jdbc.user"
            value="test" />
            <property
name="jakarta.persistence.jdbc.password"
value="test"
            />
            <property
name="jakarta.persistence.schema-
            generation.database.action"
            value="drop-and-create" />
            <property
name="hibernate.connection.autocommit"
            value="true" />
            <property
name="hibernate.allow_update_outside_transaction"
            value="true"/>
        </properties>
```

```
    </persistence-unit>
</persistence>
```

Note that we specify the **`com.mysql.cj.jdbc.Driver`** as the **`jakarta.persistence.jdbc.driver`**. Next, we set **`jdbc:mysql://localhost:3306/account`** as the database connection URL. We intend to connect to the database in our machine in port 3306, using the value **`test`** as the user and password. The property **`jakarta.persistence.schema-generation.database.action`** is set to **`drop-and-create`** on purpose to ensure Jakarta Persistence entities are dropped and created every time the application starts. This configuration is not recommended in production scenarios; we only use it here to demonstrate integration tests with MySQL databases.

Finally, we adjust the **`AccountRepository`** class to use the **`EntityManager`** to persist accounts and find them using their email addresses. Following is the code implementation that lets us persist and find **`Account`** database entities:

```
public class AccountRepository {


    @PersistenceContext
    private EntityManager entityManager;
    // Code omitted
    public void persist(Account account) {
        AccountData accountData =
convertEntityToData(account);
        entityManager.merge(accountData);
        entityManager.flush();
    }
    public Account findByEmail(String email) {
        Query query =
entityManager.createQuery("SELECT a FROM
AccountData
        a WHERE
```

```
            email = :email", AccountData.class);
            query.setParameter("email", email);
            var accountData = (AccountData)
query.getSingleResult();
            return convertDataToEntity(accountData);
        }
        // Code omitted
}
```

The **`persist`** method receives the **`Account`** object that we convert to an **`AccountData`** Jakarta Persistence entity object required to persist data into the database. The **`findByEmail`** method receives a **`String`** representing the email address used to query **`AccountData`** objects that convert to the **`Account`** type. We do this conversion because **`Account`** is the domain entity object, and **`AccountData`** is the database entity object.

We are ready to implement an integration with test Testcontainers.

Implementing an integration test with Testcontainers

We start by creating the **`EndToEndIT`** testing class with a configuration that makes MySQL containers available for our tests:

```
@Testcontainers
public class EndToEndIT {

    @Container
    public MySQLContainer<?> mySQLContainer = new
    MySQLContainer<>("mysql:8.3.0")
            .withDatabaseName("account")
            .withUsername("test")
            .withPassword("test")
            .withExposedPorts(3306)
            .withCreateContainerCmdModifier(cmd ->
cmd.withHostConfig(
```

```
                    new
HostConfig().withPortBindings(new

PortBinding(Ports.Binding.bindPort(3306), new
                    ExposedPort(3306)))
            ));
    private RegistrationService
registrationService;

    @BeforeEach
    public void init() {
        registrationService = new
RegistrationService(new
        ValidatorService(), new
AccountRepository());
    }
    // Code omitted
}
```

The **`@Testcontainers`** enables the automatic creation of containers during the JUnit 5 tests. It ensures that containers are running before testing starts. Next, we have the **`@Container`** annotation on top of the instance variable **`mySQLContainer`** that receives a **`MySQLContainer`** instance that is initialized with the **`"mysql:latest"`** Docker image tag that is pulled from the Docker Hub registry. We also use the **`withX`** methods to set database settings such as database **`name`**, **`username`**, and **`password`**. The **`withExposedPorts`** and **`withCreateContainerCmdModifier`** ensure the application can connect to the MySQL container through the 3306 port. Remember that was the port we defined in the **`persistence.xml`** file:

```
<property name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/account" />
```

Testcontainers let us create containers per test or a single container shared by all tests in the class. We control this behavior using a

static or instance variable with the **`@Container`** annotation. In our example, the **`mySQLContainer`** is an instance variable, so a new container will be created for every test method.

Once the Testcontainers configuration is done, we can write our test method, as follows:

```
@Testcontainers
public class EndToEndIT {
    // Code omitted
    @Test
    public void
givenAnActiveAccountIsProvided_thenAccountIsSuspended()
    throws Exception {
        // Arrange
        var email = "suspended@daviveira.dev";
        var password = "123456";
        var birthDate = LocalDate.of(2000, 1, 1);

        // Prepare
        var accountPayload =
getAccountPayload(email, password, birthDate);
        var activeAccount =
registrationService.register(accountPayload);

        // Pre-assert
        assertEquals(activeAccount.status(),
ACTIVE);

        // Execute
        var suspendAccount =
registrationService.suspend(email);
```

```
        // Post-assert
        assertEquals(suspendAccount.status(),
SUSPENDED);
    }
    // Code omitted
}
```

The above test checks if the account registration system can successfully suspend an account. First, it creates an active account. We confirm it by running a pre-assertion that confirms the **`ACTIVE`** status. Then, we execute the **`registrationService.suspend(email)`** and assert that the returned account is **`SUSPENDED`**. This integration test validates the account suspension use case using a real MySQL database provided by Testscontainers.

Next, we learn how to run integration tests with Maven.

Running integration tests with Maven

We use the maven-surefire-plugin to run unit tests, but for integration tests, we need the **`maven-failsafe-plugin`** configured in the **`pom.xml`** file:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.2.5</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
```

```
</plugin>
```

The command below shows how we can run the integration tests using Maven:

```
$ mvn integration-test
```

The command above produces an output similar to the one as follows:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------<
dev.davivieira:chapter04 >----------------------
[INFO] Building chapter04 1.0-SNAPSHOT
[INFO] --------------------------------[ jar
]---------------------------------
[INFO]
[INFO] --- maven-failsafe-plugin:3.2.5:integration-
test (default) @ chapter04 ---
[INFO] Using auto detected provider
org.apache.maven.surefire.junitplatform.JUnitPlatformProvid
[INFO]
[INFO]
-------------------------------------------------------
[INFO]  T E S T S
[INFO]
-------------------------------------------------------
[INFO] Running
dev.davivieira.account.service.EndToEndIT
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 15.41 s -- in
dev.davivieira.account.service.EndToEndIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
```

```
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time:  20.709 s
```

Note that the time required to run the integration tests with Testcontainers is usually higher than to run unit tests. This is because of the time spent bringing up containers.

We will now check how to compile and run the sample project accompanying this chapter.

Compiling and running the sample project

As a sample project, we have a Java application called **Account Registration**. It is the application we have been working with throughout the chapter.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter 04**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

It is also required to have Docker and Docker Compose installed because we use them to bring up a MySQL database container.

To run the application unit tests, go to the *Chapter 4* directory from the book's repository. From there, you need to execute the following command:

```
$ mvn test
```

Execute the following command to run the integration tests:

```
$ mvn integration-test
```

Execute the following command to compile the application:

```
$ mvn clean package
```

Before starting the application, execute the following command to bring up the MySQL Docker container:

```
$ docker-compose up -d
```

The command above must be executed from the *Chapter 4* root directory.

Finally, we can start the application:

```
$ java  -jar target/chapter04-1.0-SNAPSHOT-jar-with-dependencies.jar
```

```
Account[email=test@davivieira.dev, password=123456, birthDate=1980-01-01,
creationTimestamp=2024-03-25T01:29:17.256603061Z, status=ACTIVE]
```

Conclusion

When we talk about testing Java applications, we mainly talk about running unit and integration tests. Understanding what they are, their benefits, and how to employ them constitutes a fundamental skill for any Java developer. We learned in this chapter how unit tests backed by JUnit 5 can help validate application behaviors at the unit, self-contained level, of methods containing sequences of instructions, including methods with external resource calls like database access. For such methods, we learned how to mock external calls using *Mockito*. Moving ahead, we tapped into the Testcontainers capabilities to quickly provide containers for integration tests.

In the next chapter, we will examine the Java software development frameworks, starting our exploration with Spring Boot. We will cover the fundamentals of the Spring Boot framework, learn how to bootstrap a new project, implement a RESTful API with Spring Web, and persist data with Spring Data JPA.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 5
Building Production-Grade Systems with Spring Boot

Introduction

Software development frameworks constitute the cornerstone of most enterprise applications, providing the foundation for developing robust systems. Relying on features provided by a framework can also save us precious time. Aware of the benefits development frameworks can offer, we explore essential Spring Boot features to build Java enterprise applications.

Over the years, Spring Boot has become one of the most mature software development frameworks, vastly used across various industries. If you are an experienced Java developer, you have encountered at least one Spring Boot application during your career. If you are a beginner Java developer, chances are high that you will find Spring Boot applications crossing your path in your developer journey. No matter how experienced you are, knowing Spring Boot is a fundamental skill to remain relevant in the Java development market.

This chapter starts with a brief introduction to Spring fundamentals and then proceeds with a hands-on approach to developing a simple CRUD Spring Boot application from scratch with RESTful support.

Structure

The chapter covers the following topics:

- Learning Spring fundamentals
- Bootstrapping a new Spring Boot project
- Implementing a CRUD application with Spring Boot
- Compiling and running the sample project

Objectives

By the end of this chapter, you will have mastered the fundamental skills to build Spring Boot applications, handle HTTP requests through RESTful APIs, and persist data into relational databases using the Spring Data JPA. You will also grasp how Spring Boot's convention-over-configuration philosophy helps you quickly set up new applications without spending too much time on configuration details. With the knowledge in this chapter, you will have the essentials to tackle challenging Spring Boot projects.

Learning Spring fundamentals

As we embark on our journey to build a Spring Boot application, let us first understand the power of Spring and its core concepts. These are the building blocks that make Spring Boot a popular choice for Java developers.

Previously, if you wanted to build Java applications supporting transactions, security, and data persistence, you could use a framework called Java **Enterprise Edition** (**EE**). That was the official enterprise framework initially offered by *Sun Microsystems*, which Oracle acquired. Oracle continued maintaining Java EE until it decided to open source the project by handing it to the Eclipse Foundation, which renamed it Jakarta EE.

Creating enterprise applications using Java EE in its first releases did not provide the best developer experience. Although sufficient

to support the development of mission-critical applications, the Java EE specifications were not simple to use and required considerable effort from developers to have things in place. That is when, around 2002, *Rod Johnson* decided to create the Spring Framework to deliver functionalities quite similar to those offered by Java EE but without the complexity of the official framework. Effectively, the Spring Framework was conceived as an interface for Java EE, providing a simpler abstraction for Java EE features like servlets, persistence with JPA (Java Persistence API), and messaging with JMS (Java Message Service). Since then, the adoption of the Spring Framework has tremendously increased over the years due to its ease of use, time-tested stability, outstanding developer experience, and engaging community.

Spring is now considered an ecosystem of frameworks that serves the most diverse purposes in software development. Built on the backbone of the Spring Core project are other projects like Spring Data JPA, Spring Security, and, of course, Spring Boot.

The Spring Core project provides functionalities like context, dependency injection, and aspects found in most Spring Boot applications.

The Spring Boot project, in turn, takes an opinionated view from Spring maintainers to provide a framework that lets developers quickly bootstrap new applications, following a philosophy that favors convention over configuration.

Let us start our exploration by learning what a Spring context is and how we can use it to manage objects known as **beans**.

Using the Spring context to manage beans

When we decide to use Spring to develop an application, we must know that we are delegating part of the execution control and dependency handling to Spring. By execution control, we refer to the **Inversion of Control** (**IoC**) idea, where the application is not the only agent responsible for creating instances of objects it needs to perform its activities. The Java objects management responsibility can be transferred from the Java application being developed to Spring, effectively inverting the object management control from the Java application to the Spring Framework. A similar idea can be applied to dependency handling, where Spring

provides the dependencies that the application requires rather than the other way around, where the application itself needs to provide its dependencies directly.

This discussion about responsibilities is essential because it enables us to understand the purpose of the Spring context, which is used to provide objects managed by Spring. If we want our application to benefit from Spring's features, we need to consider which objects of our application must be handled by Spring. The way we make Spring manage objects is by putting them into its context. These objects managed by Spring are also known as beans. We can create beans from the classes of the application we are developing, but we can also create beans from classes of third-party libraries.

There are different approaches to creating beans. We can, for example, create beans using the **`@Bean`** annotation in conjunction with Spring configuration classes or through an XML file configuration. However, the XML approach is uncommon nowadays because it is more verbose and complicated to maintain Creating beans using Spring stereotype annotations like **`@Component`** or **`@Service`** is also possible. Next, let us learn how to create beans using the **`@Bean`** annotation.

Creating beans with the @Bean annotation

To understand the idea behind Spring context and beans, let us consider the following example:

```
class Person {
    private String name;
    // Getters and setters omitted
}

public class PersonExample {

    public static void main(String... args) {
        var person = new Person();
        person.setName("John Doe");
```

```
        System.out.println(person.getName()); //
John Doe
    }
}
```

The following steps describe how the **`Person`** and **`PersonExample`** classes are implemented:

1. Create a **`Person`** class containing the **`name`** attribute.
2. Inside the **`main`** method from the **`PersonExample`** class, we create a new **`Person`** instance by invoking its empty constructor.
3. Set the **`Person's`** name to *John Doe* and print it.

In the previous example, we created the **`Person's`** object using the class's constructor. However, there is another approach where Spring can be responsible for object creation.

Consider the following example where Spring is responsible for creating the **`Person`** object:

```
@Configuration
class PersonConfiguration {
    @Bean
    public Person person() {
        var person = new Person();
        person.setName("John Doe");
        return person;
    }
}
// Code omitted
public class PersonExample {

    public static void main(String... args) {
        var context = new
```

```
AnnotationConfigApplicationContext(PersonConfig.class);
        var person =
context.getBean(Person.class);
        System.out.println(person.getName()); //
John Doe
    }
}
```

The following steps describe how the **`PersonConfiguration`** and **`PersonExample`** classes are implemented:

1. We place the **`@Configuration`** annotation on the **`PersonConfiguration`** class. The **`@Configuration`** annotation lets us configure the Spring context using a Java class.
2. We place the **`@Bean`** annotation above the person method. This annotation tells Spring to create a new object instance based on the annotated method.
3. Inside the **`person`** method, we create a new **`Person`** instance, set its **`name`** attribute, and return it. Beans have names; by convention, a bean is named after the method name that produces the bean instance, which in our example is the **`Person`** name.
4. To use the **`Person`** bean, we need to initialize the Spring context by passing the configuration class:

```
var context = new AnnotationConfigApplicationContext(Pers
```

5. The **`AnnotationConfigApplicationContext`** lets us programmatically create a Spring context, which we can use to retrieve a bean by calling the **`getBean`** method with the bean class type:

```
var person = context.getBean(Person.class);
```

The approach described by the previous example works well when there is only one bean of the **`Person`** type in the Spring context. However, we can have trouble if we produce multiple beans of the **`Person`** type:

```
@Configuration
class PersonConfiguration {
    @Bean
    public Person person() {
        var person = new Person();
        person.setName("John Doe");
        return person;
    }
    @Bean
    public Person anotherPerson() {
        var person = new Person();
        person.setName("Mary Doe");
        return person;
    }
}
```

The second bean is defined by the **`anotherPerson`** method, which also returns a **`Person`**. We get an exception if we try to retrieve a bean by just informing the bean type to the **`getBean`** method:

```
var person = context.getBean(Person.class);

// NoUniqueBeanDefinitionException: No qualifying
bean of type 'dev.davivieira.Person' available:
expected single matching bean but found 2:
person,anotherPerson
```

We can overcome it by specifying also the bean name to the **`getBean`** method:

```
var person = context.getBean("anotherPerson",
Person.class);

System.out.println(person.getName()); // Mary Doe
```

It is also possible to override the default behavior where the method name is used as the bean name and, instead, define your own bean's name:

```
@Bean("mary")
 public Person anotherPerson() {
      var person = new Person();
      person.setName("Mary Doe");
      return person;
}
...
var person = context.getBean("mary",
Person.class);
System.out.println(person.getName()); // Mary Doe
```

In the above example, instead of relying on the default behavior where the bean name would be **`anotherPerson`**, we defined our bean name as **`mary`**. We referred to it when calling the **`getBean`** method from the Spring context.

The previous bean creation examples considered only our classes, but we can also create beans from classes we do not own:

```
@Bean
public LocalDate currentDate() {
    return LocalDate.now();
}
```

That is especially useful when you want Spring to control instances of third-party classes.

Using the **`@Bean`** annotation is not the only way to add beans into the Spring context. We can also use special Spring stereotype annotations to transform a class into a bean. Let us explore this further.

Creating beans with Spring stereotype annotations

We use Spring stereotype annotations to express classes' roles in a Spring application. This helps us quickly grasp what a class is responsible for by looking at its annotation. Next, we briefly examine the most frequently used stereotype annotations.

The @Component annotation

It is a class-level annotation representing a Spring bean object. It has the same effect as producing beans using the **`@Bean`** annotation approach. When we place the **`@Component`** annotation in a class, Spring puts it into its context and becomes responsible for creating instances of the class annotated with the **`@Component`** annotation.

The @Service annotation

Deriving from the **`@Component`** annotation, the **`@Service`** is often used on classes containing some business logic. This annotation is usually used on layered-based applications, where the service layer contains classes annotated with the **`@Service`** annotation. Spring also creates and manages **`@Service`** annotated class instances through its context.

The @Repository annotation

We use the **`@Repository`** annotation whenever a class is responsible for interacting with a database. Such an annotation became quite popular due to the repository pattern, which aggregates into a repository class logic to handle database operations for entity classes.

Using stereotype annotations

What all the stereotype annotations have in common is that they all turn a Java class into a Spring bean.

The following is an example showing how to create a Spring bean using the **`@Component`** annotation:

```
@Configuration
@ComponentScan(basePackages = "dev.davivieira")
class PersonConfiguration {
}
@Component
class Person {
    private String name;
    // Code omitted
```

```
}
public class PersonExample {
    public static void main(String... args) {
        var context = new
AnnotationConfigApplicationContext(PersonConfig.class);
        var person =
context.getBean(Person.class);
        System.out.println(person.getName()); //
null
    }
}
```

It is not enough to just place the **`@Component`** annotation on top of the **`Person`** class to make Spring detect and put it into its context. We need to tell Spring where it must look to find classes annotated with stereotype annotations. To achieve it, we use **`@ComponentScan(basePackages = "dev.davivieira")`** with the **`@Configuration`** annotation in the **`PersonConfiguration`** class. Note that the **`PersonConfiguration`** class is empty this time, as we no longer provide beans using the **`@Bean`** annotation. The **`@ComponentScan`** annotation accepts the **`basePackages`** parameter to specify the packages containing the classes Spring needs to scan.

The Person class is in the **`dev.davivieira`** package in our example. Note that in the previous example, when we call **`person.getName()`** to print the **`name`** attribute of the **`Person`** bean, it returns **`null`**, which is expected because Spring provides a **`Person`** instance by using its default empty constructor and not setting any value to class attributes like the name **`Person's`** name attribute.

When using the **`@Component`** annotation to create beans, we can set class attribute values using the **`@PostConstruct`** annotation. This annotation is not part of the Spring; it comes from the Jakarta EE framework, but Spring uses it. We use the **`@PostContruct`** annotation when we want Spring to execute some code just after the

bean object is created. The following is an example of how we can use such annotation to set the Person's name value:

```
@Component
class Person {

    private String name;

    @PostConstruct
    private void postConstruct() {
        this.name = "John Doe";
    }
    // Code omitted
}
```

The **`postConstruct`** method is called just after the **`Person`** bean instance is created.

There is also the **`@PreDestroy`** annotation that lets us execute code just before a bean instance is destroyed. It is often used to close resources like database connections.

The significant difference between beans created using stereotype annotations like **`@Component`** and those created using the **`@Bean`** annotation is that the stereotype annotations can only be used with our classes. In contrast, the **`@Bean`** annotation approach lets us create beans from classes we do not own, like those from third-party libraries.

Next, let us explore how to use beans to provide dependency injection using the **`@Autowired`** annotation.

Injecting dependencies with @Autowired

In object-oriented programming, a class can augment its capabilities by relying on behaviors and data provided by other classes. This is called composition and is a well-known alternative to inheritance. Such classes represent dependencies that need to be provided somehow to a class that depends on those dependencies. Without

dependency injection, a class has to create instances of the other classes on which it depends. Consider the following example:

```
class Skills {

    private static final Logger logger =
    Logger.getLogger(Skills.class.getName());

    void drive(String name) {
        // do something
        logger.log(Level.INFO, name+ " knows how
to drive");
    }
}
class Person {

    private Skills skills;

    public Person() {
        this.skills = new Skills();
    }
    // Code omitted
}
```

The following steps highlight some of the major points regarding the implementation of the **`Skills`** and **`Person`** classes:

1. The **`Skills`** class contains the **`drive`** method, which represents a skill a person can have.
2. The **`Person`** class depends on the **`Skill`** class. To fulfill this dependency, the **`Person`** class creates a new instance of the **`Skill`** class through its no-arguments constructor.
3. The **`Person`** is responsible for creating the **`Skill`** instance it needs to carry on with its activities.

Next, consider how the same **`Skills`** class dependency can be provided through dependency injection:

```
class Person {


    private Skills skills;
    public Person(Skills skills) {
        this.skills = skills
    }
    // Code omitted
}
```

Instead of creating the **`Skills`** object, the **`Person`** class expects to receive that object through its constructor. However, who can provide a **`Skills`** instance to the **`Person`** class? The Spring dependency injection mechanism is the answer that fills this gap. Let us see how Spring accomplishes it by first turning the **`Skills`** class into a Spring bean:

```
@Component
class Skills {


    private static final Logger logger =
    Logger.getLogger(Skills.class.getName());
    void drive(String name) {
        // do something
        logger.log(Level.INFO, name+" knows how to
drive");
    }
}
```

The **`@Component`** annotation lets Spring create and manage instances of the **`Skills`** class. Following it, we need to inject the **`Skills`** dependency into the **`Person`** class:

```
@Component
```

```
class Person {
    // Code omitted
    @Autowired
    public Person(Skills skills) {
        this.skills = skills;
    }
    public void drive () {
        skills.drive(name);
    }
    // Code omitted
}
```

The crucial element here is the **`@Autowired`** annotation placed at the **`Person's`** constructor. The **`Person`** class is a bean managed by Spring. So, when Spring creates a new instance of the **`Person`** class, it detects a constructor annotated with **`@Autowired`** that declares a dependency on the **`Skills`** object. Since **`Skills`** is also a Spring bean, Spring can get an instance of the **`Skill`** class from its context and use it to initialize the **`Person`** object.

When we get a **`Person`** bean from the Spring context, we get one that is properly initialized with its **`Skills`** bean dependency:

```
public class PersonExample {

    public static void main(String... args) {
        var context = new
AnnotationConfigApplicationContext(PersonConfig.class);
        var person =
context.getBean(Person.class);
        person.drive(); // John Doe knows how to
drive
    }
```

```
}
```

By calling the **`drive`** method from the **`Person`** bean, we can confirm it works because it relies on the behavior provided by the **`Skills`** bean.

Injecting dependencies using the **`@Autowired`** annotation with the class constructor is not Spring's only dependency injection approach. We can also inject dependencies by placing the **`@Autowired`** annotation on top of an instance attribute:

```
@Component
class Person {

    @Autowired
    private Skills skills;
}
```

Although possible, this approach is not recommended because it can make running unit tests of classes with dependencies injected directly into class attributes more difficult.

Now that we know how Spring creates and injects beans, let us explore an exciting feature that allows us to intercept methods executed by Spring beans.

Providing new application behaviors with aspects

Whenever we want to add new application behavior, the standard approach is to modify the class we want to include the new behavior. That is an invasive approach that means changing the working code. If we do not have good unit tests, such code changes have the potential to cause unwanted side effects.

We can rely on **aspect-oriented programming** (**AOP**) to decrease the risks of adding new application behaviors by modifying existing code. AOP lets us add new application behaviors without changing existing code. Spring has its own AOP framework. Let us explore how it works by first understanding some AOP principles.

Aspect

Any behavior that can be shared across different application

components is called an aspect. Aspects are also considered cross-cutting elements because they capture these shared application behaviors. Once a given behavior is implemented as an aspect, it can be used by different parts of an application. Aspects represent the code containing the behavior you want to add without changing the existing code.

Jointpoint

Application behaviors are usually represented through method executions or jointpoints from the Spring AOP perspective.

Advice

We must define when the new behavior will be executed through an aspect. For that purpose, we can rely on advice to determine if an element must be executed before or after an existing application's method.

Pointcut

The Spring AOP framework intercepts existing application methods and executes aspect code before or after them. Pointcut represents these methods intercepted by Spring.

With a fundamental understanding of AOP concepts, we are ready to explore how they work in a Spring application. Let us start by learning how to enable the aspect mechanism.

Using the Spring AOP

To get started with Spring AOP, we need to enable it in the configuration class. Let us do that using the **`PersonConfiguration`** class we used in other examples:

```
@Configuration
@ComponentScan(basePackages = "dev.davivieira")
@EnableAspectJAutoProxy
class PersonConfiguration { }
```

The **`@EnableAspectJAutoProxy`** annotation enables the aspect mechanism on Spring.

Before we implement an aspect, let us first consider the scenario

where we log the method execution of an application:

```
@Component
class Person {
    // Code omitted
    private static final Logger logger =
    Logger.getLogger(Person.class.getName());

    public void drive () {
        logger.log(Level.INFO, "Executing skill");
        skills.drive(name);
        logger.log(Level.INFO, "Skill executed
with success");
    }
    // Code omitted
}
```

To log the execution of **`drive`** method from the **`Person`** class, we need to call **`logger.log`** before and after calling **`skills.drive(name)`**. When executed, it produces an output similar to the one as follows:

```
Apr 06, 2024 9:35:41 PM dev.davivieira.Person drive
INFO: Executing skill
Apr 06, 2024 9:35:41 PM dev.davivieira.Skills drive
INFO: John Doe knows how to drive
Apr 06, 2024 9:35:41 PM dev.davivieira.Person drive
INFO: Skill executed with success
```

With Spring AOP, we can move the execution of the **`logger.log`** to an aspect class:

```
@Aspect
@Component
public class LogSkillAspect {
```

```
    private static final Logger logger =

Logger.getLogger(LogSkillAspect.class.getName());


    @Around("execution(*
dev.davivieira.Person.drive(..))")
    public void logSkill(ProceedingJoinPoint
joinPoint) throws Throwable {
        logger.log(Level.INFO, "Executing skill");
        joinPoint.proceed();
        logger.log(Level.INFO, "Skill executed
with success");
    }
}
```

We examine the **`LogSkillAspect`** implementation through the following steps:

1. We start by placing the **`@Aspect`** and **`@Component`** annotations above the **`LogSkillAspect`** class declaration. The **`@Component`** is required because the **`@Aspect`** does not make the **`LogSkillAspect`** class a Spring bean.
2. We use the Java Logging API to get a Logger object.
3. We use the **`@Around`** annotation to specify which methods should be intercepted.
4. The string **`"execution(* dev.davivieira.Person.drive(..))"`** we pass matches the **`drive`** method from the **`Person`** class. We could use the string **`"execution(* dev.davivieira.Person.*(..))"`** if we wanted to match any method of the **`Person`** class. The * character is a wildcard that can mean different things depending on where it is placed. For example, when used just after the execution first parentheses, it matches any method return type. When used after the class name, like **`Person`** in our example, it matches any method name.

5. Note the **`logSkill`** method receives a **`ProceedingJoinPoint`** as a parameter. This parameter represents the method being executed. Inside the **`logSkill`** method, we call **`logger.log(Level.INFO, "Executing skill")`** before calling **`joinPoint.proceed()`**.
6. The **`proceed`** method delegates control to the intercepted **`Person's drive`** method. After the **`drive`** method executes, **`logSkill`** calls **`logger.log(Level.INFO, "Skill executed with success")`**.

With the aspect adequately implemented, we can remove the **`logger`** from the **`Person`** class:

```
@Component
class Person {
    // Code omitted
    public void drive () {
        skills.drive(name);
    }
    // Code omitted
}
```

We get the following output when rerunning the application with the aspect adequately implemented:

```
Apr 06, 2024 10:27:37 PM
dev.davivieira.LogSkillAspect logSkill
INFO: Executing skill
Apr 06, 2024 10:27:37 PM dev.davivieira.Skills drive
INFO: John Doe knows how to drive
Apr 06, 2024 10:27:37 PM
dev.davivieira.LogSkillAspect logSkill
INFO: Skill executed with success
```

The log starts with the **`"Executing skill"`** entry, which refers to the **`dev.davivieira.LogSkillAspect`** class, followed by the **`"John Doe knows how to drive"`** entry, which refers to the **`dev.davivieira.Skills`** class, where the **`drive`** method is

executed. After the `drive` method executes, the `LogSkillAspect` class retakes control and provides the last `"Skill executed with success"` log entry.

Aspects, beans, and dependency injection comprise the building blocks of most Spring Boot applications. Now that we understand how those building blocks work, let us explore how to bootstrap a new Spring Boot project.

Bootstrapping a new Spring Boot project

As part of the Spring Framework ecosystem, Spring Boot lets developers quickly bootstrap new Spring applications. The Spring Boot framework is built following the convention-over-configuration idea. This idea is based on observing how most Spring applications are developed. Such observations allowed Spring maintainers to identify specific standards shared across those Spring applications. These standards were based on how a given Spring component would be configured. The Spring architects found out that most of the time, the configuration of those Spring components would be the same in many different Spring applications. So, they thought: why not assume this configuration as the default convention when this Spring component is used? Hence, this convention-over-configuration principle ended up being the cornerstone of the Spring Boot.

You get an application with most of its configuration already provided by Spring Boot, based on the convention of how most applications are configured. Instead of configuring the application from scratch, you only change parts of the configuration that are relevant to your project.

Let us start our exploration by learning how to initialize a new Spring Boot project.

Creating a Spring Boot project with Spring Initializr

To start a new Spring Boot project, you can rely on the Spring Initializr, which is available online on the website **https://start.spring.io/**, as an IDE plugin (for example, IntelliJ Ultimate offers it) and as a **command line interface** (**CLI**) application you can run from your machine. You can also start a new Spring Boot

project by manually specifying the Spring Boot dependencies in your Maven or Gradle project.

This book explores the CLI option, which is compatible with Windows, Mac, and Linux operating systems. Installing the Spring Initializr CLI is outside the scope of this book, but you can find instructions on how to install it at **https://docs.spring.io/spring-boot/docs/current/reference/html/cli.html**.

The following is how we can create a new Spring Boot project using the Spring Initializr CLI:

```
$ spring init --build=maven --java-version=21 --
dependencies=web sample-project

Using service at https://start.spring.io

Project extracted to '/home/m4ndr4ck/sample-project'
```

We use the **`spring init`** command to set up a new project. The following steps examine the parameters used to customize the project:

1. The **`--build`** parameter allows us to specify which build tool the project uses. Maven and Gradle are the available build tool options.
2. Next, we can use the **`--java-version`** parameter to set the project's Java version.
3. The **`--dependencies`** allow us to provide a comma-separated list of dependencies required by our project. Here, we pass only the web dependency to enable us to create RESTful applications.

Inside the sample-project directory, you will find a **`pom.xml`** with the following dependencies:

```
<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</
artifactId>

<version>3.2.4</version>

<relativePath/> <!-- lookup parent from repository
-->
```

```
</parent>
<!-- Code omitted -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
  <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
        <artifactId>spring-boot-starter-test</
artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The parent block defines the Spring Boot version, which is 3.2.4, and the **Bill Of Materials** (**BOM**) dependency identified by the **`artifactId spring-boot-starter-parent`** from where other Spring dependencies like **`spring-boot-starter-web`** and **`spring-boot-starter-test`** come from. These starter dependencies group everything related to a specific Spring Boot capability. For example, the **`spring-boot-starter-web`** dependency will correctly get all the sub-dependencies required to ensure our Spring Boot project can be used to serve RESTful HTTP requests, which includes a dependency of the embedded Tomcat server that makes our Spring Boot project runs as a standalone web application capable of receiving HTTP requests.

Although it is possible to deploy a Spring Boot project in an application server like WebLogic, most projects rely on the embedded Tomcat feature, which allows Spring Boot applications to be easily packed and executed with container technologies like Docker.

Inside the sample-project directory, we have the **`sample-`**

project/src/main/resources/application.properties file that lets us configure Spring Boot. Using the **application.yml** file to leverage the YAML syntax is also possible.

The **spring init** command used in our demonstration also creates a Java class at **sample-project/src/main/java/com/example/sampleproject/DemoApplication.java**. Use the following code:

```
@SpringBootApplication

public class DemoApplication {


 public static void main(String[] args) {

    SpringApplication.run(DemoApplication.class,
args);

}

}
```

The **@SpringBootApplication** aggregates other annotations like **@EnableAutoConfiguration** and **@ComponentScan** that automatically configure our Spring Boot application. The **SpringApplication.run(DemoApplication.class, args)** inside the **main** method brings the Spring Boot application alive. We can check it by first compiling the sample project:

```
$ ./mvnw clean package
```

This command will produce a JAR file located at **sample-project/target/sample-project-0.0.1-SNAPSHOT.jar** that we can execute to start the application:

```
$ java -jar target/sample-project-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.4)
```

```
2024-04-07T00:42:57.504+02:00  INFO 31973 --- [demo]
[           main]
c.example.sampleproject.DemoApplication  : Starting
DemoApplication v0.0.1-SNAPSHOT using Java 21.0.1
with PID 31973 (/home/m4ndr4ck/sample-project/target/
sample-project-0.0.1-SNAPSHOT.jar started by m4ndr4ck
in /home/m4ndr4ck/sample-project)

2024-04-07T00:42:57.508+02:00  INFO 31973 --- [demo]
[           main]
c.example.sampleproject.DemoApplication  : No active
profile set, falling back to 1 default profile:
"default"

2024-04-07T00:42:58.033+02:00  INFO 31973 --- [demo]
[           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat
initialized with port 8080 (http)

2024-04-07T00:42:58.041+02:00  INFO 31973 --- [demo]
[           main]
o.apache.catalina.core.StandardService   : Starting
service [Tomcat]

2024-04-07T00:42:58.041+02:00  INFO 31973 --- [demo]
[           main]
o.apache.catalina.core.StandardEngine    : Starting
Servlet engine: [Apache Tomcat/10.1.19]

2024-04-07T00:42:58.057+02:00  INFO 31973 --- [demo]
[           main] o.a.c.c.C.[Tomcat].[localhost].[/]
     : Initializing Spring embedded
WebApplicationContext

2024-04-07T00:42:58.058+02:00  INFO 31973 --- [demo]
[           main]
w.s.c.ServletWebServerApplicationContext : Root
WebApplicationContext: initialization completed in
506 ms

2024-04-07T00:42:58.239+02:00  INFO 31973 --- [demo]
[           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat
started on port 8080 (http) with context path ''

2024-04-07T00:42:58.247+02:00  INFO 31973 --- [demo]
[           main]
```

```
c.example.sampleproject.DemoApplication  : Started
DemoApplication in 1.055 seconds (process running for
1.278)
```

As we can see from the output, a Spring Boot web application runs on the 8080 port by default. We got a Spring Boot application up and running in a few steps. This the starting point for creating a new Spring Boot project.

Next, we will learn how to create a simple CRUD application with RESTful support using Spring Boot.

Implementing a CRUD application with Spring Boot

To better understand how different Spring Boot components fit together, in this section, we will implement an application that creates, retrieves, updates, and deletes persons from a database, which is fundamentally what a CRUD system does. We will also explore how to expose a RESTful API that allows us to execute the CRUD operations.

Let us start by configuring the project's dependencies.

Setting up dependencies

Using Maven as the build tool for our Spring Boot project, we define the following dependencies in the **`pom.xml`** file:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
 <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</
artifactId>
    </dependency>
```

```
    <dependency>
 <groupId>com.h2database</groupId>
 <artifactId>h2</artifactId>
 <scope>runtime</scope>
    </dependency>
</dependencies>
```

The **`spring-boot-starter-web`** dependency allows the Spring Boot application to expose RESTful HTTP endpoints. With **`spring-boot-starter-data-jpa`**, we can interact with databases using ORM technologies like Hibernate and Jakarta Persistence. The **`h2`** dependency lets us use an in-memory database while the application is running.

With the dependencies in place, we can configure the Spring Boot application.

Configuring the Spring Boot application

Let us configure our Spring Boot project using the **`application.yml`** file:

```
spring:
  datasource:
    url: jdbc:h2:mem:mydb
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    database-platform:
org.hibernate.dialect.H2Dialect
```

Leveraging the YAML syntax, we instruct Spring Boot to use the H2 in-memory database by providing the connection details through the data source configuration.

To ensure the Spring Boot application can be started, we implement the **`SpringSampleApplication`** class:

```
@SpringBootApplication
public class SpringSampleApplication {

    public static void main(String... args) {

SpringApplication.run(SpringSampleApplication.class,
args);
    }
}
```

At this stage, the Spring Boot project can already be up and running. We are ready to start implementing the application logic. Let's start with the database entity.

Defining a database entity

We want to use the Spring Boot application to manipulate the **`Person`** database entity. To do so, we first need to implement a database entity class:

```
@Entity
public class Person {

    @Id
    String email;

    String name;
    // Getters and setters omitted
}
```

We place the **`@Entity`** annotation above the **`Person`** class to make it into a Jakarta Persistence database entity. Spring Data JPA lets us easily interact with databases by using repository interfaces. Let us explore it further in the next section.

Creating a repository

To handle **`Person`** database entities, we need to implement a

repository interface:

```
@Repository

public interface PersonRepository extends
CrudRepository<Person, String> {

    Optional<Person> findByEmail(String email);

}
```

The **`CrudRepository`** is an interface provided by Spring with standard database operations like **`delete`**, **`save`**, and **`findAll`**. Extending from the **`CrudRepository`**, we can create our interface with new operations, like in the above example, by declaring the **`findByEmail`** on the **`PersonRepository`** interface. The **`CrudRepository`** is a generic type in which we need to specify the entity class - **`Person`** in our example—handled by the repository and the type used by the entity's ID, which in our case is a **`String`** because the email is the entity's ID.

Following the convention-over-configuration approach, Spring Data JPA lets us define methods like **`findByX`**, where **`X`** can be one of the entity attributes.

With the entity and its repository implemented, we can create a service responsible for handling database objects.

Implementing a service

It is a common practice to have service classes containing logic related to the database entities. Also, service classes are sometimes introduced to form a service layer that intermediates communication with repository classes from the data layer. The following code can be used to implement the **`PersonService`** class:

```
@Service

public class PersonService {


    private final PersonRepository
personRepository;
```

```
    @Autowired
    public PersonService(PersonRepository
personRepository) {
        this.personRepository = personRepository;
    }

    public Optional<Person> getPerson(String
email) {
        return
personRepository.findByEmail(email);
    }

    public void addPerson(Person person) {
        personRepository.save(person);
    }

    public void deletePerson(Person person) {
        personRepository.delete(person);
    }

    public List<Person> listAllPersons() {
        return (List<Person>)
personRepository.findAll();
    }
}
```

Spring Boot injects the **`PersonRepository`** class into the **`PersonService's`** contructor. The **`@Autowired`** annotation is not mandatory here, as Spring Boot understands that the constructor's parameters are dependencies that need to be injected. Nevertheless, we keep it here to emphasize the dependency injection activity that is taking place in the **`PersonService's`** constructor.

At this point, we have the application's entity, repository, and service classes. The only thing missing is a controller class that exposes API endpoints.

Exposing API endpoints with a controller

We must implement a controller class to define endpoints that let external clients interact with the system to allow the Spring Boot application to receive HTTP requests. We do that by implementing the **`PersonController`** class:

```
@RestController
public class PersonController {

    private final PersonService personService;

    @Autowired
    PersonController(PersonService personService)
{
        this.personService = personService;
    }

    @GetMapping("/person")
    private List<Person> getAllPersons() {
        return personService.getAllPersons();
    }

    @PostMapping("/person")
    private void addPerson(@RequestBody Person
person) {
        personService.addPerson(person);
    }

    @GetMapping("/person/{email}")
```

```
    private Person getPerson(@PathVariable String
email) throws Exception {
        return
personService.getPerson(email).orElseThrow(() ->
new
        Exception("Person not
        found"));
    }
}
```

We start by placing the **`@RestController`** annotation above the **`PersonController`** class, enabling the provision of methods that handle HTTP requests. Note we have annotations like **`@GetMapping`** and **`@PostMapping`**. We use them to define the endpoint's relative URL and the HTTP method it supports, such as **`GET`** or **`POST`**.

Note that we also use the **`@RequestBody`** annotation with the **`Person`** class. This annotation allows the Spring Boot to map a JSON payload request attributes into class attributes. If we send a JSON payload with the **`email`** and **`name`** attributes, Spring Boot will map such attributes to the **`Person`** class attributes.

We can also pass URL parameters as we did with **`@GetMapping("/person/{email}")`**, where the **`email`** parameter value is captured by the **`@PathVariable`** annotation.

The controller is the last piece of our CRUD application. Let us see how we can play with it next.

Sending HTTP requests to the Spring Boot application

There are many ways to send HTTP requests to an application. We can use tools like Insomnia, Postman, or Curl. In this book, we use Curl due to its simplicity and ability to be used through the command line.

After starting our Spring Boot application, we can send HTTP requests to see how it behaves. Following, we cover the steps for sending the requests.

1. Let us start by creating some persons:

```
curl -X POST localhost:8080/person -H 'Content-type:appli

curl -X POST localhost:8080/person -H 'Content-type:appli

curl -X POST localhost:8080/person -H 'Content-type:appli
```

2. We can confirm the `Person` was indeed created by making the following request:

```
$ curl -s  localhost:8080/person/person1@davivieira.dev |

{

  "email": "person1@davivieira.dev",

  "name": "Person 1"

}
```

   The `jq` is a command-line tool that formats JSON data to make it easier to read. If you do not have `jq` installed on your machine, you can execute only curl, but the JSON output will not be formatted.

3. Next is how we can get all persons from the database:

```
curl -s  localhost:8080/person | jq

[

  {

    "email": "person1@davivieira.dev",

    "name": "Person 1"

  },

  {

    "email": "person2@davivieira.dev",

    "name": "Person 2"

  },

  {

    "email": "person3@davivieira.dev",

    "name": "Person 3"
```

```
    }
]
```

The **/persons** endpoint returned all person entries available in the Spring Boot application database, as expected.

In this section, we learned how Spring Boot can orchestrate different application components like entity, repository, service, and controller classes to quickly provide capabilities to handle database entities and expose a RESTful API. Next, we will learn how to compile and run the sample project accompanying this chapter.

Compiling and running the sample project

The sample project from this chapter is quite similar to the CRUD application we developed in the previous session. The system enables clients to manage an individual's records from the database through a RESTful API.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2005**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

It is also required to have Curl installed because we use it to send HTTPS requests to the application.

To compile the project, go to the *Chapter 5* directory from the book's repository. From there, you need to execute the following command:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the following command:

```
$ java -jar target/chapter05-1.0-SNAPSHOT.jar
```

With the application running, you can use the following command to create a new person record:

```
$ curl -X POST localhost:8080/person -H 'Content-
type:application/json' -d '{"email":
"john.doe@davivieira.dev", "name": "John Doe"}'
```

The following command lets you retrieve an existing person from the application:

```
$ curl -s  localhost:8080/person/
john.doe@davivieira.dev
```

```
{"email":"john.doe@davivieira.dev","name":"John Doe"}
```

The output is a JSON response with personal data retrieved from the database by the Spring Boot application.

Conclusion

This chapter taught us how powerful Spring Boot can be for developing enterprise applications. Starting with Spring fundamentals, we grasped essential concepts like beans representing Java objects managed by Spring through its context. To fully tap into the benefits provided by Spring beans, we explored how the Spring dependency injection mechanism lets us inject beans into other beans. In closing the fundamentals topic, we learned about **aspect-oriented programming** (**AOP**) and how the Spring AOP lets us add new application behaviors without modifying existing code. We checked how easy it is to start a new Spring Boot application using the Spring Initializr CLI. Finally, we developed a CRUD Spring Boot application with RESTful support to understand how different components are arranged in a Spring Boot project.

In the next chapter, we continue our journey through Java frameworks by exploring Quarkus's cloud-native approach. We will learn the benefits Quarkus provides and how to kickstart a new Quarkus project that will serve as the foundation for developing a system exploring Quarkus features such as Quarkus DI, Quarkus REST, and Panache.

Chapter 6
Improving Developer Experience with Quarkus

Introduction

Designed to be a cloud-first framework, Quarkus presents an attractive alternative for creating applications to run in the cloud. Based on industry standards through specifications provided by projects like Jakarta EE and Microprofile, Quarkus helps developers build cloud-native applications by offering through its framework libraries supporting dependency injection, data persistence, RESTful APIs, and much more.

Quarkus manages to conciliate cloud-native and enterprise development practices gracefully, bringing the best of both worlds and making developers' lives easier. So, this chapter introduces Quarkus and covers some of its features most frequently used in enterprise Java applications running in the cloud.

Structure

The chapter covers the following topics:

- Assessing Quarkus benefits
- Kickstarting a new Quarkus project
- Building a CRUD app with Quakus
- Writing native applications
- Compiling and running the sample project

Objectives

By the end of this chapter, you will learn why Quarkus can be a viable framework choice for your next Java project. Once you grasp how fluid software development can be when using Quarkus, you will get why it can help boost developer productivity. You will also acquire the skills to build modern Java applications by learning the Quakus way to develop enterprise software.

Assessing Quarkus benefits

Previously, Java development was not associated with frameworks capable of empowering developers to build applications optimized to tap into the advantages provided by cloud environments. Before cloud-native development practices became as widespread as they are today, it was common to see many Java projects relying on old-fashioned application servers like WebSphere or Weblogic. Anyone who has ever worked with such technologies understands how nontrivial it is to set up a local development environment with a properly configured application server.

With the decreasing cost of computing resources, virtualization technologies like containers have become popular, allowing faster development. Developers embraced a new approach where instead of packing their Java systems to run in old-fashioned application servers, they started to pack them in Docker containers to run in Kubernetes clusters. However, it was not about just packing the application to run in a different environment. Having a Java application running in the cloud created opportunities and challenges that forced developers to think of new ways to design their applications to extract the most of what cloud environments could provide.

Positioning itself as an alternative to frameworks from an era where cloud computing did not exist, Quarkus was built from scratch as a cloud-first development framework for Java. It is cloud-first because the whole framework is designed to help developers create applications capable of benefiting from the advantages offered by cloud technologies, especially containerization. Suppose you are starting a new Java project and intend to run it in a Kubernetes cluster. In that case, Quarkus can help with features that optimize the execution of your application inside Kubernetes.

One of Quarkus's most vital points is its reliance on industry standards to provide the capabilities most enterprises need. Such standards are based on rigorous specifications determining how a given feature should work. For example, the Quarkus dependency injection mechanism is based on the Jakarta EE (formerly Java EE) **Contexts and Dependency Injection** (**CDI**) specification, which has been continuously adjusted and improved for many years. Building your application on Quarkus means you are adhering to consistent and stable industry standards, making your application more robust.

Regarding computing resource utilization, Quarkus has built-in native image support backed by GraalVM. This technology allows Java applications to be compiled into native code instead of bytecode. Native code considerably decreases the starting time of Quarkus applications, making them suitable for use cases where the time required to start an application directly impacts cloud resource costs.

Besides being cloud-first and compliant with industry standards, Quarkus is designed to enhance developer productivity and joy. Features like live coding let developers see the impact of their code changes without having to restart the application, saving precious developer time that would be otherwise spent stopping, recompiling, and starting the application again.

Quarkus is a vast framework with numerous features for many different scenarios. So, this chapter focuses only on the fundamental framework components used in most Quarkus projects. We will start by learning how simple it is to bootstrap a new Quarkus application. After we have our Quarkus app up and running, we will explore how to let Quarkus DI manage our application's objects

through the CDI. Next, we learn how Quarkus interacts with databases using Hibernate ORM and Jakarta Persistence. Finally, we explore how to create RESTful endpoints with Quarkus REST.

We will start by learning how to kickstart a new Quarkus project.

Kickstarting a new Quarkus project

Bootstrapping a Quarkus application from scratch involves setting it up with a build tool like Maven or Gradle to get the proper dependencies and provide the correct configuration to compile and run the Quarkus project. Fortunately, we do not need to do it manually because we can use Quarkus CLI to set up new projects quickly.

Quarkus CLI is available for Windows, Mac, and Linux operating systems. You can find at **https://quarkus.io/guides/cli-tooling** instructions on how to install Quarkus CLI on your machine.

Over the next steps, we will bootstrap a Quarkus project.

1. Once you have Quarkus CLI installed, the following is how you can create a new Quarkus project using the `quarkus` command:

```
$ quarkus create app my-project

Looking for the newly published extensions in registry.

-----------

applying codestarts...

java

maven

quarkus

config-properties

tooling-dockerfiles

tooling-maven-wrapper
```

```
 rest-codestart
-----------

[SUCCESS]
  quarkus project has been successfully generated in:

--> /home/m4ndr4ck/my-project

-----------

Navigate into this directory and get started: quarkus d
```

The command above creates a new Quarkus project in the `my-project` directory. By default, it uses Maven as the build tool, but Gradle and JBang are also supported. We pass the `app` option to create a Quarkus project that runs as a server application. It is also possible to create CLI applications with the `cli` option. Java is the default language, but Kotlin and Scala are also supported.

2. Once the Quarkus project is created, we can check which extensions are enabled. For that, we need to enter into the project's directory and execute the following command:

```
$ quarkus ext ls

Looking for the newly published extensions in registry.

Current Quarkus extensions installed:

✯ ArtifactId                                                E

✯ quarkus-rest                                              R

To get more information, append `--full` to your comman
```

By default, new Quarkus projects come only with the REST extension enabled. With the below command, we can list all available extensions that can be installed:

```
$ quarkus ext list --concise -i

Current Quarkus extensions installable:

✯ ArtifactId                                              Ext

✯ blaze-persistence-integration-quarkus-3                 Bla
```

```
✯ camel-quarkus-activemq                                    Cam
✯ camel-quarkus-amqp                                        Cam
✯ camel-quarkus-arangodb                                    Cam
...
...
```

The extension list is quite long, so we display only some of the first entries in the output above.

3. We can add a new extension to an existing Quarkus project by running the following command inside the project's directory:

```
$ quarkus ext add quarkus-jdbc-h2

[SUCCESS] ✅
  Extension io.quarkus:quarkus-jdbc-h2 has been install
```

The above command changes the **pom.xml** file of the Quarkus project to include the H2 dependency:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jdbc-h2</artifactId>
</dependency>
```

We use the following command to remove the H2 dependency:

```
$ quarkus ext rm quarkus-jdbc-h2

[SUCCESS] ✅
  Extension io.quarkus:quarkus-jdbc-h2 has been uninsta
```

If we recheck the **pom.xml** file, the H2 dependency will no longer be listed there.

4. Once we finish adding Quarku's project extensions, we can start the application by executing the following command:

```
$ quarkus dev
```

Executing this command will trigger the compilation and

startup in the development mode of the Quarkus application. When in the development mode, Quarkus provides a user interface at the URL **http://localhost:8080/q/dev-ui/welcome**, which is shown in the following figure:

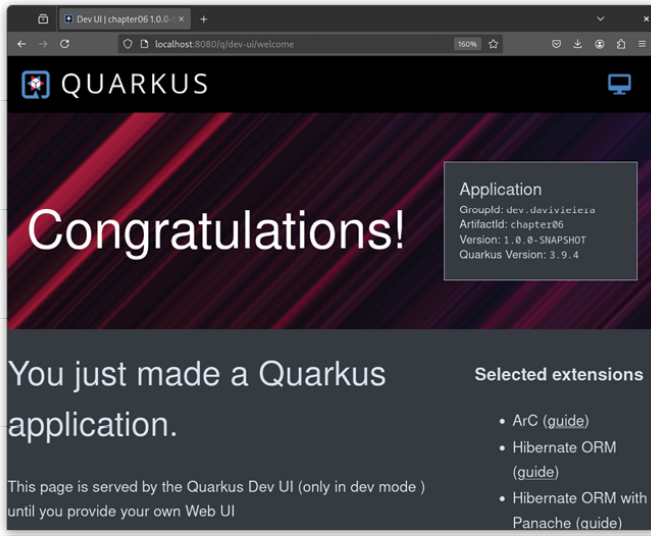

Figure 6.1: *Quarkus Dev UI*

Loading this page in your browser confirms the Quarkus application is up and running.

As part of the bootstrap process, the Quarkus CLI provides a Java class at **`src/main/java/org/acme/GreetingResource.java`** with the following code:

```
@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello from Quarkus REST";
    }
```

```
}
```

The `GreetingResource` class exposes a REST endpoint that we can test by sending an HTTP GET request to the application:

```
$ curl localhost:8080/hello
```

```
Hello from Quarkus REST
```

It shows the Quarkus application correctly handles HTTP requests.

If you are familiar with Spring Boot, you will appreciate that Quarkus eliminates the need for a bootstrap class to start the application. While it is still an option for customizing the start-up process, Quarkus is designed to start the application without it.

Now that we have the Quarkus application up and running, let us implement a CRUD project using fundamental Quarkus features like Quarkus DI, Hibernate, Panache, and Quarkus REST.

Building a CRUD app with Quakus

This section covers some of the fundamental Quarkus features for developing back-end applications that persist data and expose a REST API. After learning essential Quarkus components like Quarkus DI, Hibernate, and Panache, we will implement, using Quarkus REST, a CRUD-based account system allowing us to manage account credentials. Let us start our exploration with Quarkus DI.

Injecting dependencies with Quarkus DI

The ability to inject dependencies is an essential feature for most development frameworks, and Quarkus is no different. At the beginning of the chapter, we learned that Quarkus relies on industry specifications to provide most of its features. Hence, it implements the Jakarta **Contexts and Dependency Injection** (**CDI**) 4.0 specification, allowing developers to inject dependencies in a Quarkus application. Quarkus DI, also known as ArC, is the framework component that implements such a specification.

Compared to Spring, Quarkus achieves the exact outcome of

allowing developers to create, inject, and intercept managed beans. Spring and Quarkus differ in how the dependency injection activity takes place. To get an idea of how Quarkus injects dependencies, we need to learn about managed beans and how they are produced and injected within a Quarkus-based application.

Let us learn more about managed beans.

Managed beans

A managed bean is a Java object controlled by the framework. In Spring, managed beans live in the application context. In Quarkus, they live in what is called the container, which is the framework environment where the application is running. The managed beans' lifecycle is controlled by the container that decides when managed beans are created and destroyed. In Quarkus, we can create beans at different levels, including the class, method, and field levels. It is also possible to define the bean scope to determine how visible it will be to other beans in the application. Some of the scopes we can use in a Quarkus project are application-scoped, singleton, and request-scoped. Next, we explore how to create and inject beans using such scopes.

Application-scoped beans

Whenever we need to provide an object that should be accessible from any place in the application, we can use application-scoped beans. An application-scoped bean object is created once and lives for the entire application runtime. The following is how we can create an application-scoped bean:

```
@ApplicationScoped
public class Person {

    private String name = "John Doe";

    public String getName() {
        return name;
    }
```

```
}
```

That is a class-level application-scoped bean because we put the **`@ApplicationScoped`** annotation on top of the **`Person`** class. Following is how we can use this bean in another part of the application:

```
@Path("/person")
public class SampleApplication {

    @Inject
    Person person;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String personName() {
        return person.getName(); // John Doe
    }
}
```

The **`SampleApplication`** is annotated with **`@Path("/person")`**, making it a RESTful endpoint. It contains a **`Person`** attribute annotated with **`@Inject`**. We call injection point **`class`** attributes annotated with **`@Inject`**. When the Quarkus application starts, it tries to locate a managed bean that can be assigned to an injection point. Because we have annotated the **`Person`** class with the **`@ApplicationScoped`**, Quarkus can find a managed bean **`Person`** object and assign it to the **`person`** attribute injection point. For those from the Spring world, the **`@Inject`** annotation works similarly to the **`@Autowired`** annotation.

It is worth noting that application-scoped beans are lazy loaded, which means their instances are created only when one of their methods or attributes is invoked. In our previous example, the **`Person`** bean instance is created only when its **`getName`** method is called.

An alternative to application-scoped beans is the singleton beans. Let us check how they work.

Singleton beans

Like application-scoped beans, singleton beans are also created only once and made available for the entire application. Contrary to application-scoped beans, singleton beans are eagerly loaded, appearing when the Quarkus application starts. The following is how we can create a singleton bean:

```
@Singleton
public class Location {

    public List<String> cities =
List.of("Vancouver", "Tokyo", "Rome");

    @Produces
    List<String> countries() {
        return List.of("Canada", "Japan",
"Italy");
    }
}
```

Here, we have two beans, one created at the class level through the **`@Singleton`** annotation and the other at the method level with the **`@Produces`** annotation. The following code is how we can inject those beans:

```
@Path("/location")
public class SampleApplication {

    @Inject
    Location location;
    @Inject
    List<String> countries;

    @GET
```

```
    @Produces(MediaType.TEXT_PLAIN)
    public List<String> location() {
        return Stream
                .of(countries, location.cities)
                .flatMap(Collection::stream)
                .toList();
                // [Canada, Japan, Italy,
Vancouver, Tokyo, Rome]
    }
}
```

Consider the two injection points, represented by the **`location`** and **`country`** attributes. Upon the start of a Quarkus application, these attributes are eagerly assigned with bean instances that are compatible with their types. In this case, injecting **`Location`** would be enough to allow direct access to both the **`cities`** field and **`countries`** method from the **`Location`** singleton bean instance. However, we introduce a new bean based on the **`countries`** method to illustrate a specific use case: the possibility of having beans at the method level.

Let us explore the practical application of request-scoped beans in Quarkus. This understanding will equip us with the knowledge to effectively manage beans in real-world scenarios.

Request-scoped beans

Both application and singleton-scoped beans are accessible from any part of the system and last for as long as the Quarkus application is alive. On the other hand, request-scoped beans let us create objects available only in the context of an HTTP request. Once the request is finished, the request-scoped bean object ceases to exist. Every HTTP request the Quarkus application receives will trigger the creation of a new request-scoped bean object. Following is an example showing how to use request-scoped beans:

```
@RequestScoped
public class Account {
```

```
    private String name;
    private String email;
    private int randomId;

    @PostConstruct
    private void setAccountAttributes() {
        this.name = "John Doe";
        this.email = "john@davivieira.dev";
        this.randomId = new Random().nextInt(50);
    }

    @Override
    public String toString() {
        return "Account{" +
                "name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", randomId=" + randomId +
                '}';
    }
}
```

The **`@RequestScoped`** annotation turns the **`Account`** class into a request-scoped bean. Note that we have the **`@PostConstruct`** annotation above the **`setAccountAttributes`** method, which initializes the class attributes after creating the bean instance. Every time a new **`Account`** instance is created, the **`setAccountAttributes`** will be executed to initialize all the attributes, including the **`random`** attribute that receives a random number. The following is how we use the **`Account`** bean:

```
@Path("/account")
public class SampleApplication {
```

```
    @Inject
    Account account;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Account account() {
        return account;
        //#1 Account{name='John Doe',
email='john@davivieira.dev',
        randomId=49}
        //#2 Account{name='John Doe',
email='john@davivieira.dev',
        randomId=23}
        //#3 Account{name='John Doe',
email='john@davivieira.dev',
        randomId=27}
    }
}
```

The comments show which response we may get every time we send a GET request to **http://localhost:8080/account**. Note the **`randomId`** changes every time a new request is sent, which confirms that new **`Account`** beans are being created for each request.

In addition to the application-scoped, singleton, and request-scoped, there are also the dependent, session, and other customized scopes. Those additional scopes provide different bean behaviors that are helpful in specific situations; however, most of the time, you will be using the scopes we covered in this section.

Quarkus comes with solid support for data persistence. Let us explore it next.

Persisting data with Hibernate

Quarkus has built-in JDBC support for many different database technologies, so you can easily connect your Quarkus application to

a database. It also relies on Hibernate ORM as the Jakarta Persistence implementation. Quarkus also has a library called Panache, which significantly enhances the experience of handling Jakarta Persistence entities. So, in this section, we explore all the aspects related to establishing a database connection and handling entities. We start by learning how to make Quarkus ready to work with databases.

Setting up Quarkus to work with databases

To enable database support to an existing Quarkus project, we need to add the required extensions:

```
$ quarkus ext add quarkus-hibernate-orm quarkus-hibernate-orm-panache quarkus-jdbc-h2

[SUCCESS] ✅ Extension io.quarkus:quarkus-hibernate-orm has been installed

[SUCCESS] ✅ Extension io.quarkus:quarkus-hibernate-orm-panache has been installed

[SUCCESS] ✅ Extension io.quarkus:quarkus-jdbc-h2 has been installed
```

The **quarkus-hibernate-orm** and **quarkus-hibernate-orm-panache** extensions add Hibernate ORM support with the Quarkus Panache library. The **quarkus-jdbc-h2** extension adds the H2 in-memory JDBC driver to the Quarkus application. Once we have the database dependencies in place, we can configure the database connection and how Hibernate should behave through the **application.properties** file:

```
quarkus.datasource.jdbc.url=jdbc:h2:mem:default

quarkus.datasource.username=admin

quarkus.datasource.password=password

quarkus.hibernate-orm.database.generation=drop-and-create
```

As we use the H2 in-memory database, settings like username and password are optional because Quarkus already provides them through a default configuration. However, we need to specify the JDBC URL, which, in our case, refers to the H2 database that will

exist only during the application runtime. By setting **`quarkus.hibernate-orm.database.generation`** to **`drop-and-create`**, we instruct Quarkus to create database tables based on the Jakarta Persistence entity classes defined in the Quarkus application. When the **`drop-and-create`** option is used with ordinary databases like MySQL or PostgreSQL, it recreates existing tables at every application start. Quarkus is quite flexible regarding database configuration, providing different ways to control the database application interaction.

Having the correct database dependencies and the database connection adequately configured, we are ready to explore how Quarkus handles databases. Let us start exploring how to use Hibernate to handle database entities in a Quarkus application.

Handling database entities with EntityManager

Let us start by implementing an entity class:

```
@Entity
public class Account {

    @Id
    private String email;
    private String password;
    // Code omitted
}
```

We can implement an **`AccountRepository`** class to handle **`Account`** entities:

```
@ApplicationScoped
@Transactional
public class AccountRepository {

    @Inject
    EntityManager entityManager;
```

```
    public void createAccount(String email, String
password) {
        entityManager.persist(new Account(email,
password));
    }

    public Account getAccount(String email) {
        return entityManager.find(Account.class,
email);
    }
}
```

We turn the **`AccountRepository`** into an application-scoped bean so we can inject and use it in other application areas. This repository class relies on the **`EntityManager`** that Quarkus injects. Since we have already configured the JDBC database connection and the Hibernate ORM dependency, a valid **`EntityManager`** bean is injected by Quarkus into the **`AccountRepository`** class, making it ready to handle any database entity. Note the usage of the **`@Transactional`** annotation; we need it whenever writing operations occur in the database. The **`createAccount`** method is responsible for the database writing operation in the example above.

The following is how we can use this **`AccountRepository`** class in a Quarkus CLI application:

```
@Command(name = "SampleCLIQuarkusApp",
mixinStandardHelpOptions = true)
public class SampleCLIQuarkusApp implements
Runnable {

    @Inject
    AccountRepository accountRepository;

    @Override
```

```
    public void run() {

accountRepository.createAccount("user1@davivieira.dev",
"pass");

accountRepository.createAccount("user2@davivieira.dev",
"pass");

accountRepository.createAccount("user3@davivieira.dev",
"pass");



System.out.println(accountRepository.getAccount("user2@d

        // Account{email='user2@davivieira.dev',
password='pass'}

    }

}
```

As an alternative to using the **`EntityManager`** directly, Quarkus provides a convenient Panache library that simplifies handling database entities. Let us explore it.

Simplifying database entity handling with Panache

Built on top of Hibernate ORM and Jakarta Persistence, Panache empowers developers to handle database entities more efficiently. It relies on the active record and repository patterns, giving more flexibility to database entity mapping activities. Let us see first how Panache applies the repository pattern.

Panache with repository pattern

We have been doing the repository pattern so far in this section by declaring a repository class and using it to handle database entities. We did that already using the **`EntityManager`** in the **`AccountRepository`** class. Let us refactor that repository class to use Panache:

```
@ApplicationScoped
```

```
@Transactional
public class AccountRepository implements
PanacheRepository<Account> {

    public Account findByEmail(String email) {
        return find("email", email).firstResult();
    }
}
```

We do not need to worry about injecting an **`EntityManager`** because Panache already gives this. Note we are implementing the **`PanacheRepository`** interface. This is a requirement to turn the class into a Panache repository class. This interface has built-in operations like **`persist`**, **`findById`**, and **`delete`**, which we can use out of the box. It is also possible to define our operations as we did in the example above with the **`findByEmail`** method. The **`find`** method provided by Panache lets us query the database entity using one of its attributes. We have the **`@Transactional`** annotation because the **`persist`** method that changes the database is inherited from the **`PanacheRepository`**. Remember that whenever having database writing operations in Quarkus, we need the **`@Transactional`** annotation.

The following is what our sample application looks like with the refactored version of the **`AccountRepository`** class:

```
@Command(name = "SampleCLIQuarkusApp",
mixinStandardHelpOptions = true)
public class SampleCLIQuarkusApp implements
Runnable {

    @Inject
    AccountRepository accountRepository;

    @Override
    public void run() {
```

```
        accountRepository.persist(new
Account("user1@davivieira.dev",
        "pass"));
        accountRepository.persist(new
Account("user2@davivieira.dev",
        "pass"));
        accountRepository.persist(new
Account("user3@davivieira.dev",
        "pass"));

System.out.println(accountRepository.findByEmail("user2@
    }
}
```

Observe that we are calling the **`persist`** method from the **`AccountRepository`** class. The **`persist`** is a built-in operation provided by Panache that lets us save entities into the database.

Let us now see how we use the active record pattern to achieve the same results we achieved using the repository pattern.

Panache with active record pattern

The idea behind the active record pattern is that instead of having separated entity and repository classes, we merge these classes into a single class representing the entity itself and the database operations we can do with such an entity. To see how it works, let us refactor the **`Account`** class:

```
@Entity
public class Account extends PanacheEntityBase {

    @Id
    private String email;
    private String password;

    public static Account findByEmail(String
```

```
email) {
        return find("email", email).firstResult();
    }
    // Code omitted
}
```

You must extend from the **`PanacheEntity`** or **`PanacheEntityBase`** abstract classes to apply the active record pattern. We are extending from **`PanacheEntityBase`** because our entity has the **`@Id`** annotation. Otherwise, Panache would handle the unique identification of the entity through **`PanacheEntity`**. Whether we are extending from **`PanacheEntity`** or **`PanacheEntityBase`**, the class extending it will inherit a set of built-in database operations like **`persist`** and **`delete`**. Similar to what we did with the repository pattern approach when applying the active record pattern with Panache, we can define our customized database operations inside the entity class like we did when defining the **`findByEmail`** method in the **`Account`** class.

The following is how we can handle **`Account`** entities using the active record pattern approach:

```
@Command(name = "SampleCLIQuarkusApp",
mixinStandardHelpOptions = true)
public class SampleCLIQuarkusApp implements
Runnable {


    @Override
    @Transactional
    public void run() {
        new Account("user1@davivieira.dev",
"pass").persist();
        new Account("user2@davivieira.dev",
"pass").persist();
        new Account("user3@davivieira.dev",
"pass").persist();
```

```
System.out.println(Account.findByEmail("user2@davivieira
    }
}
```

The **`@Transactional`** annotation is placed above the **`run`** method because, inside it, we have database writing operations triggered when we call **`persist`** after creating the **`Account`** object. With the active record pattern approach, we concentrate everything on the **`Account`** class, which establishes the database table mapping and provides customized database operations like **`findByEmail`**.

Deciding between the repository and active record patterns is something that the needs of your project will dictate.

Having covered the fundamental aspects of how Quarkus deals with databases, let us explore how to build an API with Quarkus.

Implementing an API with Quarkus REST

As with most Quarkus features, the Quarkus support for REST APIs is based on the Jakarta REST specification. The extension Quarkus REST fully implements the Jakarta specification, allowing developers to implement reactive and non-reactive endpoints. The Quarkus REST also supports JSON payloads through the Jackson or JSON-B libraries.

In this section, we will walk you through the creation of a practical Quarkus REST API. This API is capable of creating, retrieving, and deleting data from a database. We will use examples based on the **`Account`** database entity, which we have been working on in previous sections.

Next, we will look at the steps to implement the API:

1. Before we jump into the API implementation, let us first recap the **`Account`** entity class structure using the following code:

```
@Entity
public class Account {

    @Id
```

```
    private String email;

    private String password;
    // Code omitted
}
```

2. To allow handling **`Account`** entities, we have the following **`AccountRepository`** class:

```
@ApplicationScoped
public class AccountRepository implements PanacheRep

    public Account findByEmail(String email) {
        return find("email", email).firstResult();
    }
    public void deleteByEmail(String email) {
        delete("email", email);
    }
}
```

   Note that we have a new method called **`deleteByEmail`** that deletes **`Account`** entities based on the email address.

   With entity and repository classes adequately implemented, we can implement REST endpoints to allow changes to the database.

3. The following is how we can start the implementation of the **`AccountEndpoint`** class:

```
@Path("/account")
public class AccountEndpoint {

    @Inject
    AccountRepository accountRepository;
   // Code omitted
```

```
}
```

We start by placing the **`@Path("/account")`** annotation above the class name. By doing that, we establish a URL path that will be part of all endpoints defined inside the **`AccountEndpoint`** class.

4. We first implement the endpoint that allows creating new accounts:

```
@Path("/account")
public class AccountEndpoint {

    @Inject
    AccountRepository accountRepository;

    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public void create(Account account) {
        accountRepository.persist(account);
    }
    // Code omitted
}
```

HTTP requests that create resources should be handled as POST requests, so we have the **`@POST`** annotation in the **`create`** method. The **`@Transactional`** annotation is also present because this endpoint triggers a database writing operation that must be done inside a transaction. With the **`@Consumer`** and **`@Produces`** annotations, we can determine which media type this endpoint consumes and produces: **`MediaType.APPLICATION_JSON`** for both.

5. Moving on, we implement the endpoints that allow us to retrieve accounts:

```
@Path("/account")
public class AccountEndpoint {

    @Inject
    AccountRepository accountRepository;


    // Code omitted
    @Path("/{email}")
    @GET
    public Account get(@PathParam("email") String em
        return accountRepository.findByEmail(email);
    }

    @Path("/all")
    @GET
    public List<Account> getAll() {
        return accountRepository.listAll();
    }
    // Code omitted
}
```

The endpoint defined by the **`get`** method receives GET requests from the **`/account/{email}`** path. The **`{email}`** is a path parameter mapped to the endpoint method parameter at **`@PathParam("email") String email`**. We also define an additional GET endpoint at **`/account/all`** that retrieves all existing accounts.

6. Finally, we define an endpoint to delete accounts:

```
@Path("/account")
public class AccountEndpoint {
```

```
        @Inject
        AccountRepository accountRepository;


        // Code omitted
        @Path("/{email}")
        @Transactional
        @DELETE
        public void delete(@PathParam("email") String em
            accountRepository.deleteByEmail(email);
        }
        // Code omitted
    }
```

The endpoint defined by the **`delete`** method receives HTTP DELETE requests at **`/account/{email}`** that use the email address to locate and delete **`Account`** entities from the database. The **`@Transactional`** annotation is required because deletion is a writing database operation.

Once the Quarkus application is up and running, it is time to get hands-on with the API. We can interact with it by sending various requests.

We send POST requests to create new accounts:

```
$ curl  -H "Content-Type: application/json" -XPOST --
data
'{"email":"user1@davivieira.dev","password":"123"}'
localhost:8080/account

$ curl  -H "Content-Type: application/json" -XPOST --
data
'{"email":"user2@davivieira.dev","password":"123"}'
localhost:8080/account

$ curl  -H "Content-Type: application/json" -XPOST --
data
```

```
'{"email":"user3@davivieira.dev","password":"123"}'
localhost:8080/account
```

We can send the following GET request to retrieve a specific account:

```
$ curl -s  localhost:8080/account/
user1@davivieira.dev| jq
{
  "email": "user1@davivieira.dev",
  "password": "123"
}
```

The following is how we can retrieve all accounts:

```
$ curl -s  localhost:8080/account/all| jq
[
  {
    "email": "user1@davivieira.dev",
    "password": "123"
  },
  {
    "email": "user2@davivieira.dev",
    "password": "123"
  },
  {
    "email": "user3@davivieira.dev",
    "password": "123"
  }
]
```

To delete an account, we need to send an HTTP DELETE request:

```
curl -XDELETE  localhost:8080/account/
user2@davivieira.dev
```

If we send a new request to get all accounts, we can confirm that one of the accounts no longer exists in the database:

```
$ curl -s  localhost:8080/account/all| jq
[
  {
    "email": "user1@davivieira.dev",
    "password": "123"
  },
  {
    "email": "user3@davivieira.dev",
    "password": "123"
  }
]
```

Having working REST endpoints in a Quarkus application does not take much. The framework lets us easily express how the application endpoints should behave through intuitive annotations.

Let us explore next how to write native applications using Quarkus.

Writing native applications

A typical Java application is based on classes compiled into bytecode that runs inside a **Java Virtual Machine** (**JVM**). The JVM is the software we install in the operating system we want to run the Java application. Once compiled as bytecode, Java classes can be executed on different operating systems like Windows, Mac, and Linux, as long as those systems have a JVM installed on them. The JVM provides benefits like garbage collection and optimizations that let us efficiently execute Java applications. However, such benefits come with a price. A JVM running and performing all the warm-up activities required to execute a Java application consumes computing resources, especially memory.

Before a Java application becomes ready to operate and execute its tasks, the JVM must spend time processing and optimizing the application's bytecode. In most use cases, waiting for the application to start and allocating the necessary memory to run it inside a JVM is fine. However, there are scenarios where speeding up the application startup and using as little memory as possible is

crucial. The usage of Java applications for one-time executions, like those offered by function-as-service solutions such as the **Amazon Web Services (AWS)** Lambda, is one of the use cases where Java applications with smaller memory footprint provide benefits because those function-as-service solutions charge costs based on how much memory the system uses and how much time the system needs to execute its operations. Reducing the amount of memory a Java application uses and the time required to start such an application is possible. By doing so, we can save money, especially when running many sort-lived Java applications is necessary.

Let us explore a technology called native image that enables us to create memory-optimized versions of Java applications.

Introducing the native image

GraalVM is the **Java Development Kit** (**JDK**) that provides a technology called native image. This technology relies on the ahead-of-time compilation technique to compile a Java application into a standalone executable that can be executed directly in the operating system without having a **Java Virtual Machine** (**JVM**) installed. An ordinary Java application is compiled into bytecode that executes inside the JVM. GraalVM, on the other hand, compiles a Java application into native code that executes directly in the targeted operating system.

Quarkus relies on GraalVM to provide support for native image compilation. Next, we will learn how to compile Quarkus applications into native images.

Creating a native executable with Quarkus

On a Maven-based project, we can configure a profile in the **`pom.xml`** file that lets us compile a native executable of the Quarkus application:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 https://
maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"

         xmlns:xsi="http://www.w3.org/2001/
```

```
XMLSchema-instance">
<!-- Code omitted -->
<profiles>
    <profile>
        <id>native</id>
        <activation>
            <property>
                <name>native</name>
            </property>
        </activation>
        <properties>
            <skipITs>false</skipITs>
            <quarkus.native.enabled>true</
quarkus.native.enabled>
        </properties>
    </profile>
</profiles>
<!-- Code omitted -->
</project>
```

Both the profile ID and name are defined as native. We use the profile name later when executing the **`mvn`** command to compile the native executable. The native compilation is activated through the **`quarkus.native.enabled`** property set as true. After creating the Maven profile, we can create a native executable by executing the following command:

```
$  mvn clean package -Pnative
...
Produced artifacts:
 /project/chapter06-1.0.0-SNAPSHOT-runner
(executable)
 /project/chapter06-1.0.0-SNAPSHOT-runner-build-
```

```
output-stats.json (build_info)

...
```

Instead of producing a JAR file, the command above creates a native executable called **chapter06-1.0.0-SNAPSHOT-runner** that we can execute by issuing the following command from within the project's root directory:

```
$ ./target/chapter06-1.0.0-SNAPSHOT-runner

__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2024-09-08 21:47:13,318 INFO  [io.quarkus] (main)
chapter06 1.0.0-SNAPSHOT native (powered by Quarkus
3.9.4) started in 0.016s. Listening on:
http://0.0.0.0:8080

2024-09-08 21:47:13,318 INFO  [io.quarkus] (main)
Profile prod activated.

2024-09-08 21:47:13,318 INFO  [io.quarkus] (main)
Installed features: [agroal, cdi, hibernate-orm,
hibernate-orm-panache, jdbc-h2, narayana-jta, rest,
rest-jackson, smallrye-context-propagation, vertx]
```

Notice that the first logged line mentions that the Quarkus application runs in the native mode.

To wrap up what we have learned in this chapter, in the next section, we run the Account application and send sample requests to ensure it is working as expected.

Compiling and running the sample project

This chapter's sample project is based on the examples we have worked with during the previous sections. It is a Quarkus CRUD application that manages account data from a database.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2006.**

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine. To test the application endpoints, you must have curl or any other HTTP client of your preference. The **jq** command line tool is optional but helps format the JSON output generated by curl. The steps are as follows:

1. Execute the following command to compile the application:

```
$ mvn clean package
```

2. After compiling and having generated the JAR file, we can run the following command to start the Quarkus application:

```
$ java -jar target/chapter06-1.0.0-SNAPSHOT-runner.jar

__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/

2024-04-21 18:43:26,586 INFO  [io.quarkus] (main) chapt

2024-04-21 18:43:26,587 INFO  [io.quarkus] (main) Profi

2024-04-21 18:43:26,588 INFO  [io.quarkus] (main) Insta
```

3. Once the Quarkus application is running, we test it by sending HTTP requests. The following is how we can create a new account:

```
$ curl -H "Content-Type: application/json" --data '{"em
```

4. To confirm the account was created, we can send the following request:

```
$ curl -s localhost:8080/account/user1@davivieira.dev |

{

  "email": "user1@davivieira.dev",

  "password": "123"

}
```

The JSON response confirms the account was created when we sent the first request.

In the previous steps, we compiled, started, and tested the sample project by sending HTTP requests to create and retrieve an account.

Conclusion

Although Quarkus is a new framework compared to Spring, it is evolving quickly and has an engaging community. Quarkus is built on and leverages solid industry standards like those provided by Jakarta EE, a set of specifications for enterprise Java applications, and Microprofile, a set of specifications for microservices. These standards ensure that Quarkus is a viable choice for developing enterprise cloud-native applications. As we can see in this chapter, starting a new Quarkus project is just a matter of running a single command with the Quarkus CLI that creates a fully working Quarkus application. We learned how to inject dependencies using Quarkus DI. Dealing with databases is also easy, with extensive support for different database technologies and the pleasant Panache library that lets us conveniently handle Jakarta Persistence entities. We closed this chapter by exploring how simple creating an API with Quarkus REST is.

Quarkus strives to provide the best developer experience by simplifying things that take precious time. With Quarkus, developers can focus more on developing their solution than on the technical details that are taken care of by Quarkus, required to enable their solution. This includes features like live coding, which allows developers to see changes in their code immediately without having to restart the application, and fast startup times, which enable quick iteration and testing of code changes.

In the next chapter, we look further into Java enterprise development by learning how to create enterprise applications using the Jakarta EE. We will learn the Jakarta EE specification structure and how it defines standards that ensure the development of robust and reliable applications. We will also explore MicroProfile, a specification derived from the Jakarta EE that enables the development of microservices.

Chapter 7

Building Enterprise Applications with Jakarta EE and MicroProfile

Introduction

Developing enterprise-grade applications in Java involves following the standards and best practices that contribute to the robustness, stability, reliability, and maintainability requirements that often appear when building mission-critical applications. Such requirements can be met by relying on the specifications provided by the Jakarta EE, a project that prescribes how fundamental aspects of enterprise applications, like persistence, dependency injection, transactions, security, and more, should work. Although extensive in its coverage of the things enterprise application requires, the Jakarta EE does not contain specifications that support the development of lightweight cloud-native applications based on highly distributed architectures like microservices. To fill this gap,

the MicroProfile specification helps developers build modern enterprise applications that are better prepared for cloud-native environments.

This chapter teaches us how powerful Jakarta EE and MicroProfile are when combined to build modern enterprise Java applications.

Structure

The chapter covers the following topics:

- Overviewing Jarkarta EE
- Starting a new Jakarta EE project
- Building an enterprise application with Jakarta EE
- Adding microservices and cloud-native support with MicroProfile
- Compiling and running the sample project

Objectives

By the end of this chapter, you will be able to develop cloud-native enterprise applications using the Jakarta EE and MicroProfile specifications. This chapter shows you how the technologies derived from both specifications support the development of applications that harness the time-tested enterprise features provided by Jakarta while also tapping into the cloud-native development capabilities offered by MicroProfile.

Overviewing Jarkarta EE

Jakarta EE is a continuation of a project officially launched in 1999 under the name of **Java 2 Enterprise Edition** (**J2EE**). The vision behind this project was to provide a set of specifications to support the development of Java enterprise applications. These specifications would target technologies like databases, messaging, and web protocols such as HTTP and WebSockets, which were common in enterprise projects. Instead of reinventing the wheel by defining how to deal with those technologies, developers could rely on the standards provided by the J2EE specifications. Relying on the specifications would also grant some flexibility and vendor lock-

in protection because multiple vendors offered implementations of the same J2EE specifications.

Sun Microsystems, later acquired by *Oracle*, was responsible for the first versions of the J2EE specifications. In 2006, the project was renamed to Java EE until 2020, when *Oracle* decided to turn Java EE into an open-source project by giving its governance to the Eclipse Foundation, which renamed it to Jakarta EE. Oracle decided to give up on Java EE because it could not catch up on the innovations and features provided by other frameworks like Spring. Many developers regarded Java EE as too complex and less productive compared to its alternatives, which would deliver the same and even more functionalities more straightforwardly.

However, even with its reputation as complex and heavyweight, which was to a certain extent lost after the Java EE 5 version that introduced the annotation-based configuration as an alternative to the XML-based one, Java EE found strong adoption across many industries, such as banking and telecommunication, that relied on the Java enterprise to enable their most critical operations. The specifications provided the stability and robustness that big corporations needed to ensure the health of their businesses.

Many things have changed since the first J2EE/Java EE version until its latest incarnation, the Jakarta EE. Some specifications were removed because they no longer make sense today, and other specifications evolved to reflect the needs of modern software development. Although new frameworks and ways to develop enterprise Java software have appeared, some principles and ideas from Jakarta EE remain relevant and are still in use today.

Jakarta EE proposes a multitier architecture for developing enterprise applications. Let us explore what multitiered applications mean further.

Designing multitiered applications

An enterprise system is often composed of different components that complement each other to provide valuable functionalities. Such components are grouped into tiers according to their responsibility in the enterprise system. In the following section, we examine the Jakarta EE tiers.

Client tier

The client tier is where all the enterprise system clients live. A client can be a **user interface** (**UI**) served through a web browser or desktop application that interacts with the enterprise system. Other applications that are not UIs can also act as clients of the enterprise system. The main characteristic of the client-tier components is that they trigger behaviors in the enterprise system by making requests to it. A client is technology-agnostic; it can be developed in Java or any other technology.

Web tier

An enterprise system may offer a web application that renders HTML pages accessible through a web browser. Although not so common today, where most front-end development is client-side, support for server-side front-end applications is also part of the Jakarta EE specification. The **Jakarta Server Pages** (**JSP**) and **Jakarta Server Faces** (**JSF**) specifications, built on top of the Jakarta Servlet specification, are the technologies we find on the web tier that enable enterprise system components to serve web resources.

Business tier

Business rules represent the most critical component of an enterprise system. Whatever business problem an enterprise system aims to solve, the business tier has components containing the business logic responsible for solving it. That is where Jakarta EE components like Jakarta Persistence entities and stateless, stateful, and message-driven beans, also known as enterprise beans, are used to solve business problems.

Enterprise information system tier

Jakarta EE enterprise applications depend on external systems like databases, mainframes, **enterprise resource planning** (**ERP**), and any other system that provides data to fulfill the enterprise application requirements. These external systems comprise the enterprise information system tier.

The following figure illustrates how the tiers relate to each other:

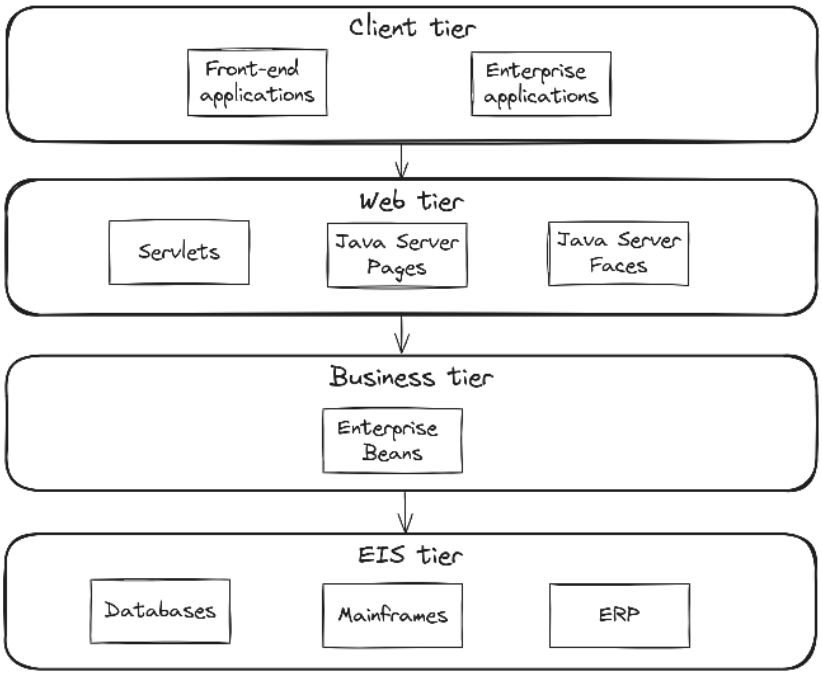

Figure 7.1: *Jakarta EE tiers*

Note that the communication flow starts with the client and then goes through the web, business, and the EIS tier. Employing all tiers is not mandatory in a Jakarta EE project. You can have an enterprise system that does not contain any web component, so the web tier would not exist in such a system.

Jakarta EE is a collection of specifications governing the development of enterprise software in Java. Let us further explore these specifications.

Exploring Jakarta EE specifications

Jakarta EE is a set of specifications for developing enterprise applications in Java. The Jakarta EE project does not provide the implementations for those specifications. It is up to the Jakarta EE vendors to implement them. Developers use the specification to build their applications and can choose which Jakarta EE vendor best suits their needs. Jakarta EE implementation is provided through Java libraries that implement the specification interfaces.

Those libraries are shipped together with the application server offered by a Jakarta EE vendor that implements the specifications or relies on third parties that implement some of the specifications. Oracle WebLogic, IBM WebSphere, Payara, and Eclipse Glassfish are some of the Jakarta EE vendors in the market.

All the Jakarta EE individual specifications are grouped into the Jakarta EE Platform and Profile specifications. Let us examine the purpose of each specification further.

Jakarta EE Platform specification

We have seen previously that Jakarta EE projects are based on a multitier architecture where different application components interact across the client, web, business, and **enterprise information system** (**EIS**) tiers. Components from the web and business tiers run in containers, a runtime environment provided by a Jakarta EE server. There are web containers responsible for executing web resources like **Java Server Page** (**JSP**) and **Enterprise Java Bean** (**EJB**) containers that execute business logic code. Components from the client and EIS tiers usually run outside a Jakarta EE server and interact with components from the web and business tiers. Running outside a Jakarta EE server, we may have client systems making calls to enterprise applications and databases providing data to such applications.

The Jakarta EE Platform specification establishes what is required from a platform aiming to host Jakarta EE applications. To comply with the specification, an application server providing a Jakarta EE platform must meet security, network, transaction, persistence, and other requirements. The specification also defines how technologies provided by different specifications can be integrated. That is helpful because it gives developers a standard for properly employing different specifications.

The following is a figure representing all the technologies covered by the Jakarta EE Platform specification:

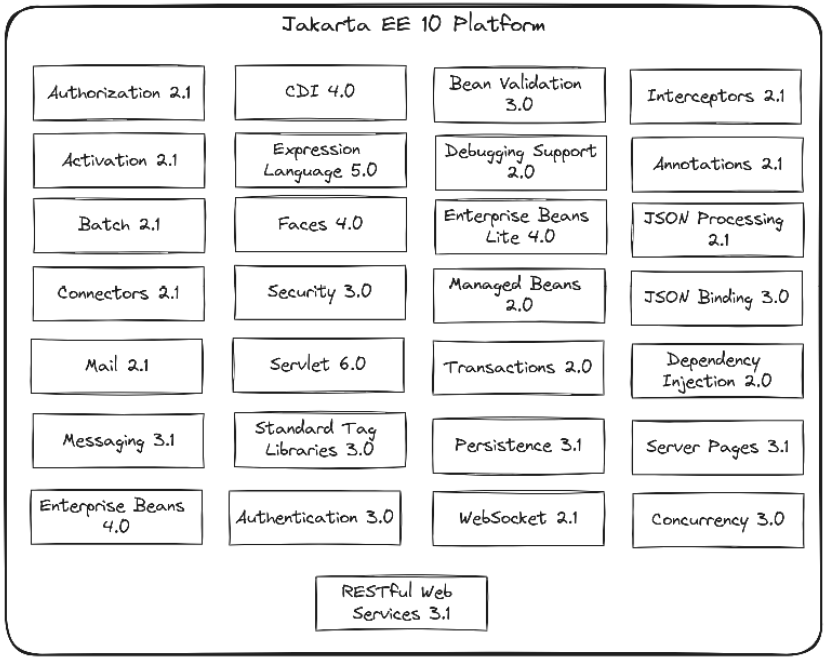

Figure 7.2: *Jakarta EE 10 Platform specification*

In terms of scope, the Jakarta EE Platform specification covers all aspects related to how a Jakarta EE project is structured, where and how it should run, and its integration with different systems and technologies.

Jakarta EE Web Profile specification

Jakarta EE Web Profile specification was the first profile specification created. Its primary purpose is to group only specifications related to the development of web applications. For example, the Jakarta Messaging is part of the Jakarta EE Platform specification, but not the Jakarta EE Web Profile specification. This means that platforms targeting the Web Profile do not need to provide a runtime environment that fulfills all the same requirements as the main Jakarta EE Platform specification, contributing to the development of smaller, leaner enterprise applications. In the following figure, we can see all the technologies that are part of the Web Profile:

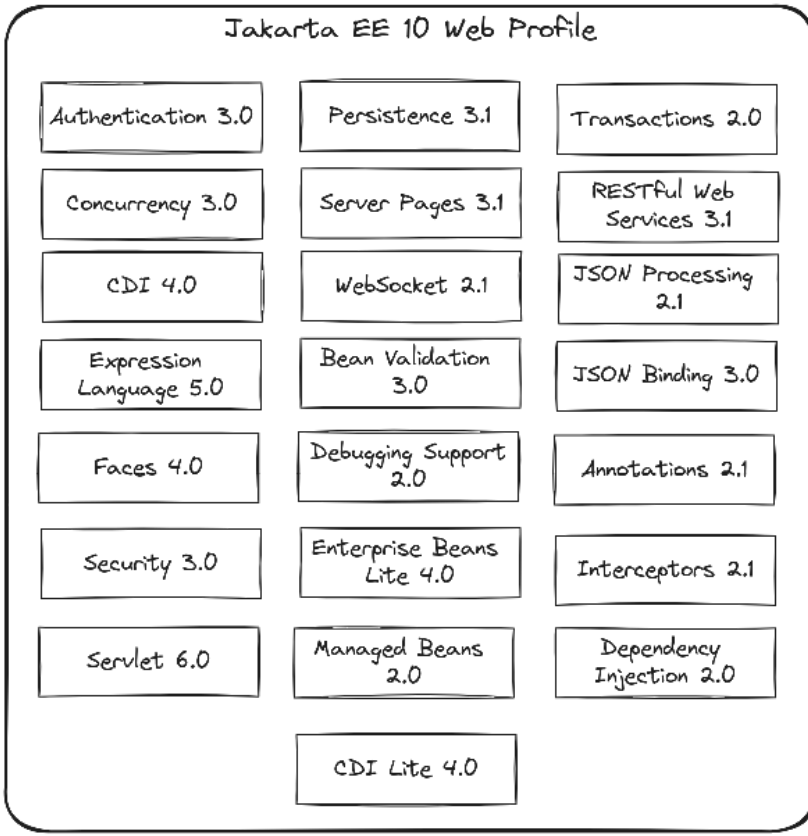

Figure 7.3: *Jakarta EE 10 Web Profile specification*

Note that some specification components from the Platform are not shown here.

Jakarta EE Core Profile specification

Created after the Platform and Web Profile specifications, the Jakarta EE Core Profile specification came out to support the development of modern cloud applications. Instead of bringing many specifications, the Core Profile relies on small specifications targeting microservices development. Projects requiring faster startup time and a smaller memory footprint can benefit from the Core Profile. A figure illustrating how the Core Profile is composed is as follows:

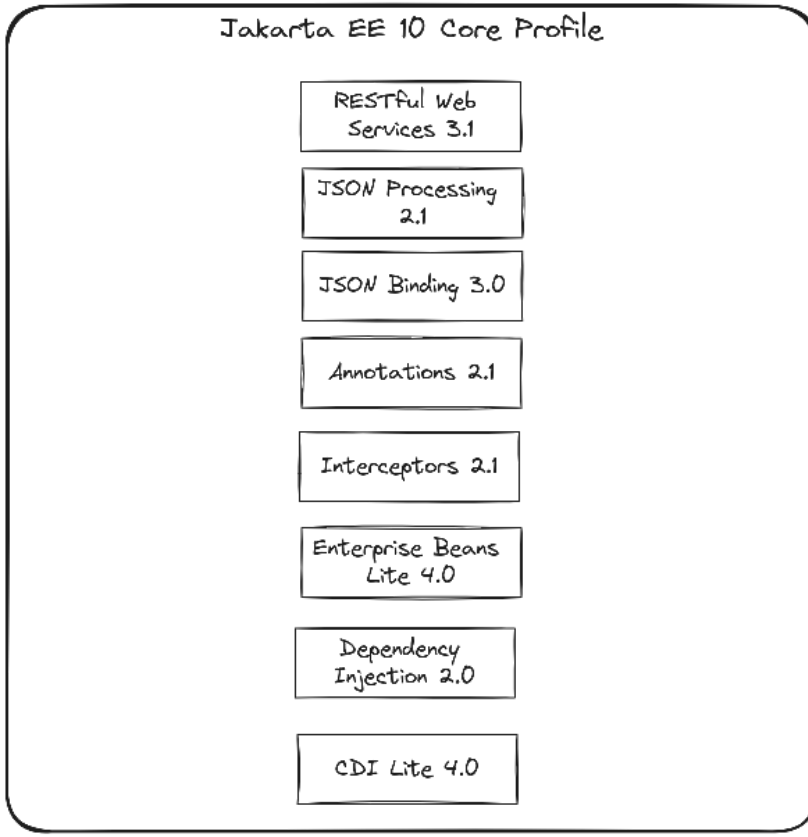

Figure 7.4: *Jakarta EE 10 Core Profile specification*

Note that only some specifications are used here. Anything that is not considered strictly necessary for the development of cloud applications is removed.

Packing, deploying, and running Jakarta EE applications

Jakarta EE applications can be packed as **Java Archive** (**JAR**), **Web Archive** (**WAR**), or **Enterprise Archive** (**EAR**). Although they have different file name extensions, such as .jar, .war, and `.ear`, all these file types share the same internal file structure based on the `.jar` file. Next, we explore the purpose of each one of these files.

Java Archive

We use `.jar` files to pack classes containing enterprise beans,

which constitute the enterprise application's business rules code.

Web Archive

Classes representing web components like servlets and other web resources, including HTML pages and images, are packaged in a WAR file. JAR files containing enterprise beans can also be bundled into a WAR file. So, in this packing structure, we can have both web and enterprise components packed into a single WAR file.

Enterprise Archive

It is possible to put all the related modules of an enterprise application into an EAR file, which allows a packing structure based on JAR files containing enterprise beans and WAR files containing web components.

Once you have packaged your enterprise application, you can deploy it into a compatible server. Fully compliant Jakarta EE projects must run on certified Jakarta EE servers, which adhere to one of the Jakarta EE Platform or Profile specifications. Applications servers like Oracle WebLogic, RedHat JBoss, and Payara are examples of certified Jakarta EE servers. It is also possible to deploy WAR and JAR files into non-certified servers like Tomcat, which does not implement all the required Jakarta EE specifications but provides enough capabilities to host enterprise Java applications.

Jakarta EE continues evolving to remain relevant as a solid platform for developing enterprise applications. In this section, we covered only the surface of this vast project, which offers a standard for creating enterprise Java applications through its specifications. Let us now explore another ramification of Jakarta EE: MicroProfile, a project that targets microservices development.

Introducing MicroProfile

Over the years, we have seen significant changes in the approaches to developing Java enterprise applications. With decreased computing costs and the popularization of cloud computing technologies like containers, developers started to think in different ways to design enterprise systems. Instead of developing a single monolith application to run in heavyweight application servers,

developers are now exploring distributed architectures based on smaller applications, the so-called microservices, that run on containers orchestrated by Kubernetes.

Organizations seeking to improve their ability to respond quickly to customer needs have widely adopted distributed architectures, like microservices. The main argument is that breaking a monolith system into smaller applications, such as microservices, helps to tackle the maintainability and scalability issues when a monolith system becomes too big. New challenges arise for the Java developer aspiring to tap into the benefits of cloud-native applications based on the microservices architecture. It is essential to understand how Java applications behave when running inside containers. Knowing how to implement monitoring and observability capabilities becomes a critical development activity to ensure all components of a distributed system are running as expected. It is also fundamental to know how the components of a distributed system communicate with each other.

It is not trivial to ensure that a Java distributed system based on multiple microservices applies the techniques and technologies to leverage all the benefits provided by cloud environments. So, MicroProfile proposes, through its specifications, a standard for developers aiming to create Java microservices that run in cloud runtimes based on container technologies like Docker and Kubernetes.

MicroProfile is similar to Jakarta EE in that it prescribes how to do things in a Java enterprise system. However, it differs by providing a set of specifications explicitly tailored for designing enterprise applications using cloud-native development techniques and technologies.

To better understand how MicroProfile works, let us explore its specifications further.

Exploring MicroProfile specifications

To guide the development of Java microservices, the MicroProfile comprises two specification sets: the Jakarta Core Profile specification and the MicroProfile specification. Next, we check all the specifications of the MicroProfile 6.1 release.

Jakarta EE Core Profile specifications

MicroProfile relies on the Jakarta EE Core Profile specifications as the foundation for the development of cloud-native applications, listed as follows:

- Jakarta RESTful Web Services 3.1
- Jakarta JSON Processing 2.1
- Jakarta JSON Binding 3.0
- Jakarta Interceptors 2.1
- Jakarta Enterprise Beans Lite 4.0
- Jakarta Dependency Injection 2.0
- Jakarta CDI Lite 4.0

The Core Profile specifications constitute the backbone of any Java enterprise application running in the cloud. We can use the Jakarta RESTful Web Services 3.1 specification to construct API endpoints capable of handling JSON payloads. The Jakarta JSON Processing 2.1 and Jakarta JSON Binding 3.0 specifications provide JSON support. The Jakarta Dependency Injection 2.0 and Jakarta CDI Lite 4.0 specifications offer a robust mechanism for handling object dependencies in an enterprise application.

Complementing the Jakarta EE Core Profile specifications, we have the MicroProfile specifications.

MicroProfile specifications

A typical Java microservice application may require monitoring, observability, configuration, security, and other essential capabilities. The MicroProfile covers the following capabilities with a set of specifications for cloud-native development practices:

- MicroProfile Telemetry 1.1
- MicroProfile OpenAPI 3.1
- MicroProfile Rest Client 3.0
- MicroProfile Config 3.1
- MicroProfile Fault Tolerance 4.0
- MicroProfile Metrics 5.1

- MicroProfile JWT Authentication 2.1
- MicroProfile Health 4.0

Troubleshooting errors is one of the challenges when using microservices architecture because to understand why a request failed, a developer may need to check the log of multiple microservices involved in the failing request. The MicroProfile Telemetry 1.1 provides advanced observability capabilities with spans and traces, elements that help us to understand the flow of requests crossing different applications. With MicroProfile Rest Client 3.0, we can make microservices communicate with one another. With MicroProfile Health 4.0, we explore the approaches to notify external agents about the health of a Java application.

The MicroProfile specifications govern the development aspects of Java enterprise applications running in cloud-native environments. Vendors like Open Liberty, Quarkus, and Payara implement the MicroProfile specifications.

After covering the Jakarta EE and MicroProfile specifications, let us learn how to use them to build Java enterprise applications.

Starting a new Jakarta EE project

A convenient way to start a new Jakarta EE project is by going to the **http://start.jakarta.ee** website. There, we can customize the project's settings by defining things like the Jakarta EE version, the Jakarta EE profile, the Java SE version, and other options. It is also possible to start a new project using Maven's archetype of a minimal Jakarta EE application, like shown in the following Maven command example:

```
$ mvn archetype:generate –
DarchetypeGroupId=org.eclipse.starter –
DarchetypeArtifactId=jakartaee10-minimal –
DarchetypeVersion=1.1.0 –DgroupId=dev.davivieira –
DartifactId=enterpriseapp –Dprofile=web-api –
Dversion=1.0.0-SNAPSHOT –DinteractiveMode=false
```

The parameters **archetypeGroupId**, **archetypeArtifactId**, **archetypeVersion** specify the archetype we want to use to generate the Maven project. We are using the archetype provided by the Eclispse foundation for new Jakarta EE projects. The other

parameters specify the configuration of the Jakarta EE project we are generating. Note that we are using the **`web-api`** profile, which makes it a project that supports the web application features, like exposing RESTful endpoints, from the Jakarta EE specification.

After executing the Maven command mentioned in the previous example, we will have the skeleton project in the **`enterpriseapp`** directory. The code in this directory will serve as the basis for building an enterprise application with Jakarta EE, which we will explore next.

Building an enterprise application with Jakarta EE

As the starting point of a Jakarta EE enterprise application that is ready to receive HTTP requests, we can extend the **`Application`** class provided by the Jakarta RESTful Web Services specification. The example below shows how we can extend the **`Application`** class:

```
@ApplicationPath("")
public class ApplicationConfig extends Application
{


}
```

The **`ApplicationConfig`** from the example above is the same one produced by the Maven command that created the initial Jakarta EE project. The **`@ApplicationPath`** annotation lets us configure the root path that will precede all RESTful endpoints our application provides. We can create a new RESTful endpoint by implementing the **`SampleResource`** class:

```
@Path("sample")
public class SampleResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sample() {
        return "Sample data for the Jakarta EE
application";
```

```
    }
}
```

The **`"sample"`** path defined here through the **`@Path`** annotation will be appended to the path provided by the **`@ApplicationPath`** defined in the **`ApplicationConfig`** class. The **`@GET`** and **`@Produces`** annotations come from the Jakarta RESTful Web Services specification. We use these annotations to expose an endpoint that receives HTTP GET requests and produces plain text responses.

To run the enterprise application, we must provide a compatible Jakarta EE runtime—an application server that implements the Jakarta EE specification. We can accomplish this using the WildFly application server. We can configure it on the project's pom.xml file by adding the following Maven plugin:

```
<plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-maven-plugin</artifactId>
    <version>5.0.1.Final</version>
    <executions>
        <execution>
            <phase>install</phase>
            <goals>
                <goal>deploy</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

The plugin above must be placed within the plugins section of the pom.xml file. Once WildFly is adequately configured, we can start the application by running the following command:

```
$ mvn clean package wildfly:run
```

We can confirm the application is working by accessing the URL

http://localhost:8080/enterpriseapp/sample in the browser or executing the following command:

```
$ curl -X GET http://localhost:8080/enterpriseapp/
sample
```

```
Sample data for the Jakarta EE application
```

We get a plain text response by sending a GET request to **http://localhost:8080/enterpriseapp/sample**.

Next, we explore how to combine Jakarta EE and MicroProfile to build enterprise applications that support cloud-native and microservices capabilities.

Adding microservices and cloud-native support with MicroProfile

In this section, we explore how to use Jakarta EE and MicroProfile specifications to build a Java enterprise application supporting all microservices and cloud-native capabilities provided by MicroProfile. We start by learning how to set up an initial Maven project with the correct dependencies, then proceed to implement a simple license management application that illustrates some of the Jakarta EE and MicroProfile capabilities.

Setting up the project

To start a new MicroProfile project, we can generate a skeleton Maven or Gradle project using the project generator at **https://start.microprofile.io/** or manually specify the dependencies required by our project. Let us proceed with the manual approach because it gives us a better understanding of the required dependencies. The following is how the **`pom.xml`** file of a MicroProfile project using Maven should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    <!-- code omitted -->
    <dependencies>
        <dependency>
            <groupId>jakarta.platform</groupId>
```

```
            <artifactId>jakarta.jakartaee-api</
artifactId>
            <version>10.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.eclipse.microprofile</
groupId>
            <artifactId>microprofile</artifactId>
            <version>6.1</version>
            <type>pom</type>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <!-- code omitted -->
</project>
```

The first dependency **`jakarta.jakartaee-api`** enables support for the Jakarta EE specifications, while the dependency **`microprofile`** brings all MicroProfile specifications to the project. Note that these dependencies do not represent the implementation of the specifications. The specification vendors provide the implementations and the runtime server where the enterprise Java application will run. We use Payara as the MicroProfile vendor for our license management application.

Continuing with the Maven project's setup, we configure the compiling and packaging plugins:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    <!-- code omitted -->
    <build>
        <finalName>license-management</finalName>
```

```
        <plugins>
            <plugin>

<groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-
plugin</artifactId>
                <version>3.11.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</
artifactId>
                <version>3.4.0</version>
                <configuration>
                    <failOnMissingWebXml>false</
failOnMissingWebXml>
                </configuration>
            </plugin>
        <!-- code omitted -->
        </plugins>
    </build>
</project>
```

We use the **`maven-compiler-plugin`** to compile the Java source files from our MicroProfile project. The **`maven-war-plugin`** lets us package the application files into a deployable WAR file.

Finally, we configure our MicroProfile project to run using the Payara runtime:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    <!-- code omitted -->
    <build>
```

```
        <finalName>license-management</finalName>
        <plugins>
        <!-- code omitted -->
                <plugin>
                    <groupId>org.codehaus.cargo</
groupId>
                    <artifactId>cargo-maven3-
plugin</artifactId>
                    <version>1.10.11</version>
                    <configuration>
                        <container>
                            <containerId>payara</
containerId>
                            <artifactInstaller>

                                <groupId>

fish.payara.distributions
                                </groupId>

<artifactId>payara</artifactId>

<version>6.2024.1</version>
                            </artifactInstaller>
                        </container>
                    </configuration>
                </plugin>
        </plugins>
    </build>
</project>
```

The **cargo-maven3-plugin** configured with the **payara** dependency allows us to run our MicroProfile application on a Payara server that is provided as a dependency of the Maven project.

After properly setting up the Maven project with the correct dependencies and build configurations, we can start development using the Jakarta EE and MicroProfile specifications. Let us begin defining a persistent data source for our license management application.

Defining a data source

The license management application relies on an H2 in-memory database to store data. We can configure this database by first creating a **web.xml** file in the **src/main/webapp/WEB-INF** directory of the MicroProfile project:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/
javaee
              http://xmlns.jcp.org/xml/ns/javaee/
web-app_3_1.xsd"
version="3.1" >
<data-source>
 <name>java:global/h2-db</name>
 <class-name>org.h2.jdbcx.JdbcDataSource</class-
name>
 <url>jdbc:h2:mem:test</url>
</data-source>
</web-app>
```

The **name** component defines the data source name the

MicroProfile application uses to establish a connection with the database. The **`class-name`** component specifies that this is an H2 data source. The **`url`** component contains the JDBC URL expressing the connection to an in-memory H2 database called test. Having a data source declaration inside a MicroProfile application's **`web.xml`** file allows this data source configuration to be deployed into the Payara server when the MicroProfile application is also deployed.

After defining the data source in the **`web.xml`** file, we need to configure the MicroProfile application to use it. We do that through the **`persistence.xml`** file in the **`src/main/resources/META-INF`** directory:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence

             xmlns="https://jakarta.ee/xml/ns/
persistence"

             xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"

             xsi:schemaLocation="https://
jakarta.ee/xml/ns/persistence

             https://jakarta.ee/xml/ns/
persistence/persistence_3_0.xsd"

             version="3.0">

    <persistence-unit name="H2DB">

        <jta-data-source>java:global/h2-db</jta-
data-source>

        <properties>

            <property
name="jakarta.persistence.schema-

            generation.database.action"

            value="drop-and-create"/>

        </properties>

    </persistence-unit>

</persistence>
```

By providing the persistence.xml file, we enable our MicroProfile application to use Jakarta Persistence. Note that the value **`java:global/h2-db`** used in the **`jta-data-source`** configuration component is the same value from the data source definition of the **`web.xml`** file. We set the property **`jakarta.persistence.schema-generation.database.action`** to **`drop-and-create`** to ensure our MicroProfile application creates tables in the H2 database. These tables are created based on the Jakarta Persistence entities implemented by the MicroProfile application.

After configuring the data source, we can start implementing the business logic of our license management application. Let us start our implementation by defining a Jakarta Persistence entity.

Implementing a Jakarta Persistence entity

The license management application enables users to create, delete, or update software licenses. To store license data in the database, let us create a Jakarta Persistence entity class:

```
@Entity
public class License {

    @Id
    @GeneratedValue(strategy =
GenerationType.SEQUENCE)
    long id;

    String name;

    LocalDate startDate;

    LocalDate endDate;

    boolean isExpired;
```

```
    // Code omitted //
}
```

We place the **`@Entity`** annotation to make this class a valid Jakarta Persistence entity. All entities require an ID, which is provided by declaring the id attribute with the **`@Id`** and **`@GeneratedValue`** annotations. The **`@GeneratedValue`** annotation is configured with the **`GenerationType.SEQUENCE`** strategy that automatically generates IDs when persisting new entities to the database. Let us create a repository class responsible for handling license database entities.

Implementing a repository with the EntityManager

Still relying on the Jakarta Persistence specification, we implement a repository class using the **`EntiyManager`**:

```
@ApplicationScoped
@Transactional
public class LicenseRepository {

    @PersistenceContext
    private EntityManager entityManager;
    public void persist(License license) {
        entityManager.persist(license);
    }
    public List<License> findAllLicenses() {
        return (List<License>) entityManager
                .createQuery("SELECT license from
License license")
                .getResultList();
    }
}
```

Coming from the Jakarta CDI specification, we have the **`@ApplicationScoped`** annotation to ensure the creation of one

managed bean instance of the **`LicenseRepository`** class. The **`@PersistenceContext`** annotation placed above the **`entityManager`** field relies on the configuration provided by the **`persistence.xml`** we created earlier. This repository class contains a method that persists **`License`** entities and another that retrieves all **`License`** entities from the database.

The persistence context represents the entities we can persist into the database. When the Java application starts, the Jakarta entities are mapped and put into the persistence context. Every persistence context is associated with an entity manager, allowing entities to be handled from such a context.

Implementing a service class as a Jakarta CDI managed bean

Service classes are usually implemented to apply business logic and handle application behaviors that may depend on external data sources. When dealing with business logic that depends on the database, the service class can have a direct dependency on the repository class responsible for handling database entities:

```
@ApplicationScoped
public class LicenseService {

    @Inject
    private LicenseRepository licenseRepository;

    public void createLicense(License license) {
        licenseRepository.persist(license);
    }

    public List<License> getAllLicenses() {
        return
licenseRepository.findAllLicenses();
    }
}
```

We turn the **`LicenseService`** class into a managed bean by using the **`@ApplicationScoped`** annotation. Inside the **`LicenseService`** class, we inject the **`LicenseRepository`** that we use to create and retrieve licenses.

Having implemented the entity, repository, and service classes using Jakarta EE specifications, let us define an endpoint class using MicroProfile and Jakarta EE specifications.

Building API with Jakarta EE and MicroProfile

We can combine Jakarta EE and MicroProfile specifications to implement a well-documented RESTful API. Let us start by defining the API base application endpoint path:

```
@ApplicationPath("api")

@OpenAPIDefinition(

        info = @Info(title = "License Management",
version = "1.0.0")

)

public class LicenseApplication extends
Application {

}
```

The **`@ApplicationPath`** annotation comes from Jakarta EE, while the **`@OpenAPIDefinition`** comes from MicroProfile. With the **`@ApplicationPath`** annotation, we define a base endpoint path that will be appended as a path component of other endpoints created for this application. The **`@OpenAPIDefinition`** lets us document the API by defining its title and version. When the application starts, the information provided by annotations like the **`@OpenAPIDefinition`** is used to generate the API documentation based on the OpenAPI specification.

Next, we start implementing the **`LicenseEndpoint`** class:

```
@Path("/license")

@Tag(name = "License API", description = "It
allows managing licenses")

public class LicenseEndpoint {
```

```
    @Inject
    private LicenseService licenseService;
    // Code omitted
}
```

The **`@Path`** annotation from the Jakarta RESTful Web Services specification is crucial as it allows us to define the endpoint path. It is worth noting that the base application path is **`/api`**, defined in the **`LicenseApplication`** class, which is then prepended to the **`/license`** path used here in the **`LicenseEndpoint`** class, resulting in the **`/api/license`** path. The **`@Tag`** annotation from the MicroProfile OpenAPI specification is essential for generating API documentation.

Continuing with the **`LicenseEndpoint`** implementation, we implement an endpoint allowing the creation of new licenses:

```
@Path("/license")
@Tag(name = "License API", description = "It
allows managing licenses")
public class LicenseEndpoint {
    // Code omitted
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(summary = "It creates a license",
description = "A new
    license is created and
    persisted into the database")
    @APIResponses(value = {
            @APIResponse(
                    responseCode = "200",
                    description = "A new license
has been successfully
                    created"
```

```
            )
    })
    public void createLicense(License license) {
        licenseService.createLicense(license);
    }
    // Code omitted
}
```

The **`@POST`** annotation creates an endpoint path accessible at **`/api/license`** through HTTP POST requests.

The **`@POST`**, **`@Consumes`**, and **`@Produces`** come from the Jakarta RESTful Web Services specification. Note that this endpoint consumes and produces JSON data. The Jakarta JSON Processing and Jakarta JSON Binding specifications provide JSON support. The remaining annotations come from the MicroProfile OpenAPI specification and allow us to provide detailed information that will be used to document this endpoint in the API documentation.

We finish by implementing an endpoint that retrieves all available licenses:

```
@Path("/license")
@Tag(name = "License API", description = "It
allows managing licenses")
public class LicenseEndpoint {
    // Code omitted
    @GET
    @Operation(summary = "It retrieves all
licenses", description = "It
    returns all non-expired
    and expired licenses")
    @APIResponses(value = {
            @APIResponse(
                    responseCode = "200",
```

```
                    description = "List of
licenses retrieved
                    successfully",
                    content = @Content(
                            mediaType =
"application/json",
                            schema =
@Schema(implementation = List.class,
                            type =
SchemaType.ARRAY)
                    )
            )
    })
    public List<License> getAllLicenses() {
        return licenseService.getAllLicenses();
    }
    // Code omitted
}
```

We place the **`@GET`** annotation to create a new endpoint path accessible at **`/api/license`** through HTTP GET requests.

We need to add the following dependency to the **`pom.xml`** file to enable the OpenAPI UI that lets us see the project's API documentation in a web browser:

```
<dependency>
    <groupId>org.microprofile-ext.openapi-ext</
groupId>
    <artifactId>openapi-ui</artifactId>
    <version>2.0.0</version>
    <scope>runtime</scope>
</dependency>
```

To ensure the OpenAPI annotations we placed when implementing

the API will be used to generate the API documentation, we need to create the **microprofile-config.properties** file in the **src/main/resources/META-INF** directory:

openapi.ui.title=License Management

mp.openapi.scan=true

The **microprofile-config.properties** allows us to configure the MicroProfile project's components, such as the OpenAPI. The **openapi.ui.title** option sets the OpenAPI UI title to License Management. We set **mp.openapi.scan** to **true** to ensure classes containing OpenAPI annotations are used to generate the API documentation.

We use OpenAPI because we want to provide a standardized API documentation describing the possible ways one can interact with the license system. In the following sessions, we will compile the license system and see how the OpenAPI user interface looks in the web browser.

Using MicroProfile Health to implement health checks

We can use health check mechanisms to determine, for example, if the application's database connection is working or if the application is consuming too much memory or CPU. In the following example, we implement the **LicenseHealthCheck** class that demonstrates how we can employ health checks in a MicroProfile project:

```
@ApplicationScoped
public class LicenseHealthCheck {

    @Produces
    @Liveness
    HealthCheck checkMemoryUsage() {
        return () ->
HealthCheckResponse.named("memory-
        usage").status( true).build();
    }
```

```
    @Produces
    @Readiness
    HealthCheck checkCpuUsage() {
        return () ->
HealthCheckResponse.named("cpu-
        usage").status(true).build();
    }
}
```

We implement the **`LicenseHealthCheck`** class as a managed bean through the **`@ApplicationScoped`** class-level annotation. The **`@Produces`** method-level annotation is used with the **`checkMemoryUsage`** and **`checkCpuUsage`** methods because they produce managed beans of type **`HealthCheck`** containing health information we collect from the application. On the **`checkMemoryUsage`** method, we have the **`@Liveness`** annotation that lets third-party services know if the MicroProfile application is running correctly. The **`@Readiness`** annotation used with the **`checkCpuUsage`** method lets third-party services know if the MicroProfile application is ready to receive requests.

Third-party services are often represented through Kubernetes cluster components responsible for periodically sending liveness and readiness probes to check the application's health. If Kubernetes detects something wrong when performing liveness and readiness health checks, it can trigger remedial actions, like restarting the Pod where the MicroProfile application is running in the Kubernetes cluster.

In the next section, we will learn how to compile and run the license management application.

Compiling and running the sample project

In the previous section, we implemented a simple license management application using Jakarta EE and MicroProfile. In this section, we explore how to compile and run this project.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-**

World-Projects/tree/main/Chapter%2007.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

Execute the following command to compile and run the application:

```
$ mvn clean package cargo:run
```

It takes time to compile and deploy the application into the embedded Payara server. You should see an output as follows when the command finishes its execution:

```
[INFO] Resolved container artifact
org.codehaus.cargo:cargo-core-container-
payara:jar:1.10.11 for container payara

[INFO] Parsed GlassFish version = [6.11.2]

[INFO] Payara 6.11.2 starting...

[INFO] Waiting for cargo-domain to start ....

[INFO] Successfully started the domain : cargo-domain

[INFO] domain  Location: /home/m4ndr4ck/IdeaProjects/
Java-Real-World-Projects/Chapter 07/target/cargo/
configurations/payara/cargo-domain

[INFO] Log File: /home/m4ndr4ck/IdeaProjects/Java-
Real-World-Projects/Chapter 07/target/cargo/
configurations/payara/cargo-domain/logs/server.log

[INFO] Admin Port: 4848

[INFO] Command start-domain executed successfully.

[INFO] Payara 6.11.2 started on port [8080]

[INFO] Press Ctrl-C to stop the container...
```

Following is how you can send requests to create new licenses:

```
$ curl -v -H "Content-Type: application/json" --data
'{"name":"Premium License", "startDate":
"2024-01-10", "startDate": "2025-01-18"}'
localhost:8080/license-management/api/license

$ curl -v -H "Content-Type: application/json" --data
'{"name":"Trial License", "startDate": "2024-01-10",
"endDate": "2025-01-18"}'  localhost:8080/license-
management/api/license
```

You can retrieve all licenses by using the following command:

```
$ curl -s localhost:8080/license-management/api/
license |jq
[
  {
    "expired": false,
    "id": 1,
    "name": "Premium License",
    "startDate": "2025-01-18"
  },
  {
    "endDate": "2025-01-18",
    "expired": false,
    "id": 2,
    "name": "Trial License",
    "startDate": "2024-01-10"
  }
]
```

It is possible to check the application's health by sending the following request:

```
$ curl -s http://localhost:8080/health | jq
{
  "status": "UP",
  "checks": [
    {
      "name": "cpu-usage",
      "status": "UP",
      "data": {}
    },
    {
```

```
        "name": "memory-usage",
        "status": "UP",
        "data": {}
    }
  ]
}
```

You can access the OpenAPI UI at **http://localhost:8080/license-management/api/openapi-ui/index.html**, the page will be as follows:

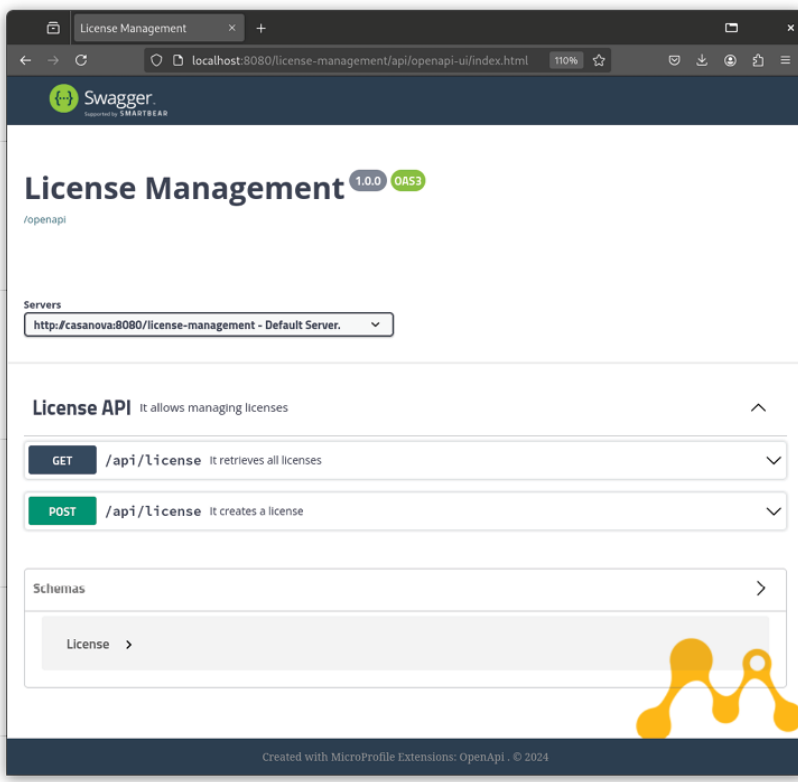

Figure 7.5: *OpenAPI UI*

The OpenAPI UI is generated from the OpenAPI documentation annotations used when implementing the API endpoints of the MicroProfile project.

Conclusion

This chapter explored Jakarta EE and how its Platform, Web Profile, and Core Profile specifications can be used to develop Java enterprise applications. While comparing the differences between the different specifications, we learned that the Platform specification includes all individual Jakarta EE specifications: the Web Profile specification, which targets enterprise applications requiring web components, and the Core Profile specification, suited for applications running in cloud-native environments. We also learned how the Jakarta EE Core Profile specification complements the MicroProfile specification to provide a set of specifications to support the development of Java microservices. After overviewing the Jakarta EE and MicroProfile specifications, we learned how to use them by developing the license management application.

The next chapter covers the techniques and technologies for deploying and running Java applications in cloud-native environments. We will learn about Kubernetes, its architecture, and how to deploy and run containerized Java applications in a Kubernetes cluster.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 8

Running Your Application in Cloud-Native Environments

Introduction

Gone are the days when we used to deploy Java applications in heavyweight application servers running on expensive dedicated hardware. Nowadays, it is far more common to see Java systems running in Kubernetes clusters offered by major cloud providers like **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**), or Microsoft Azure. By delegating most of their computing infrastructure operations to cloud providers, organizations can allocate their time to activities that have more potential to generate profit or reduce costs. The Java developer working in an organization that has its infrastructure on the cloud must know how to tap into the benefits of running Java applications in cloud-native environments. That is why this chapter explores good practices for deploying and running Java applications using technologies like Docker and Kubernetes.

Structure

The chapter covers the following topics:

- Understanding container technologies
- Introducing Kubernetes
- Dockerizing a Spring Boot, Quarkus, and Jakarta EE application
- Deploying Docker-based applications on Kubernetes
- Compiling and running the sample project

Objectives

By the end of this chapter, you will know how to package Java applications into Docker images and deploy them in a Kubernetes cluster. To serve as a solid foundation for your practical skills, you will also acquire fundamental knowledge about cloud technologies like virtualization and containers, allowing you to use such technologies properly according to your project's requirements.

Understanding container technologies

Although industries such as banking, insurance, and governmental organizations, to name a few, still rely on traditional application servers to run Java applications, most new Java projects today run in containers. Containers are not new technology. However, they matured to the point that they became an attractive technology for developers seeking better ways to deploy and run their applications.

To understand containers, we must first understand virtualization because containers are just one of the possible ways to virtualize computer resources. Virtualization comes from the desire to run software in an environment that wholly or partially represents a real computer. There can be many motivations to run applications in virtualized environments. For example, Linux users may employ virtualization to use software available only to Windows. Tools like Wine emulate Windows in Linux environments and can solve most compatibility problems. Still, sometimes, there is no way to run a specific software other than in the operating system it was originally designed for.

Another virtualization use case is running legacy applications that work only in old operating systems. Running old operating systems on new hardware is sometimes impossible due to a lack of driver support and other issues. Virtualization helps those who must run critical legacy systems that cannot be easily rewritten to run on modern platforms. We can also use virtualization as a mechanism to make an application portable. The virtualized system can provide all the dependencies an application needs to run correctly. So, instead of delivering only the application, we can provide it bundled with the virtualized environment where it runs.

Going next, we explore virtualization methods and technologies, including container-based virtualization.

Introducing virtualization

Virtualization is the core technology enabling cloud-native environments. What we call cloud computing nowadays is only possible due to the ability to virtualize computing resources. Knowledge of fundamental virtualization concepts can help us better decide how to run Java applications in cloud environments. Let us start exploring the full virtualization concept.

Full virtualization

Full virtualization is the technique of running software in an environment that reproduces a real computer's behaviors and instructions. It allows the complete virtualization of computer resources like CPU and memory. All things provided by computer hardware are virtualized, enabling the execution of software entirely unaware it is executing in a virtualized environment.

A host machine can provide full virtualization with support for hardware-assisted virtualization technologies like Intel VT-x or AMD-V. The host machine runs hypervisor software responsible for creating virtual machine instances, also known as guests of the host machine. KVM, VMware, and VirtualBox are some of the hypervisors that provide full virtualization. When a machine is fully virtualized, it lets us install an operating system that is different from the operating system running on the host machine. For example, a Linux host machine can have Windows guest machines.

Cloud providers rely on full virtualization technologies to provide

virtual servers as an **Infrastructure-as-a-Service** (**IaaS**) solution. These virtual servers run on physical servers and are managed by hypervisor software.

When organizations started to move their infrastructure to the cloud, virtual servers allowed those organizations to keep running their legacy and modern applications. An important feature of virtual servers is the flexibility to adjust computing resources like CPU and memory according to user demand. Organizations moving to the cloud would no longer struggle with over—or under-provisioning computing resources through physical hardware because these resources could now be easily managed through the virtualization hypervisor.

Full virtualization is excellent for running any application. Modern virtualization technologies make the performance of applications running in virtualized servers practically the same as if they were running in bare metal servers. However, it comes at a cost. Full virtualization is a costly way to virtualize software execution because it requires virtualizing all computer components. There is a cheaper alternative to full virtualization called paravirtualization. Let us explore it next.

Paravirtualization

Partial virtualization, also known as paravirtualization, is possible when the host machine executes some instructions of the virtual guest machine. This implies that the virtualized operating system needs to be aware that it is running in a virtual environment. The paravirtualization technique relies on the cooperation of both host and guest machines in identifying which instructions are better executed by the virtualized hardware and which others the real hardware better execute. Such collaboration between host and guest machines enhances performance in the paravirtualized environment.

A paravirtualization hypervisor requires a guest virtual machine running an operating system that can communicate with the hypervisor. So, the operating system needs to be modified or provide special drivers in order to be compatible with paravirtualized environments. That can be a limitation if you want to virtually run applications in an operating system that does not

support paravirtualization.

Paravirtualization is cheaper than full virtualization but still implies significant costs because it consumes valuable computing resources to provide a functional virtualized environment. *Xen* and *Hyper-V* are some of the available paravirtualization technologies cloud providers may use to offer virtualized servers and other cloud solutions backed by paravirtualization.

Full virtualization and paravirtualization techniques are extensively used to run Java applications. Such Java systems can run in application servers executing in virtualized environments. Kubernetes cluster nodes are provisioned using virtual servers. So, these virtualization techniques are awesome for running Java applications. Although they can make Java applications portable, it may be cumbersome to package and distribute those applications bundled into full or paravirtual machine images.

Let us explore a virtualization technique that allows virtualizing the structure of an operating system, which enables a convenient way to package applications and their dependencies into a single, leaner, virtualized environment.

Container-based virtualization

Also known as OS-level virtualization, container-based virtualization consists of virtualizing components of an operating system. In this approach, the guest virtualized environment shares the same kernel used by the host machine. This container virtualization does not go as deep as full virtualization or paravirtualization, which virtualizes hardware instructions to provide the virtual environment. We call containers the virtual environments offered by container-based virtualization technologies.

Container technology has been around for quite a while through solutions like OpenVZ, **Linux Containers** (**LXC**), and Docker. It is based on two essential Linux kernel features: cgroups and namespaces. These features let applications run as isolated processes within an operating system. The namespace kernel mechanism allows the creation of an isolated environment in the hosting operating system. From the application perspective, an environment provided by a Linux namespace looks like a real operating system

containing its own file system and process tree. Processes running in one namespace cannot see processes from other namespaces. Computing resources like CPU and memory are managed by the cgroups kernel feature, which is responsible for allocating computing resources to containers controlled by the host operating system.

A container can run as one or more isolated processes. Processes running inside a container behave as if they were running in a real machine.

Before the advent of Docker, containers were not widely embraced by developers. The earlier container technologies lacked a straightforward method to package and distribute environment dependencies with application binaries. However, Docker revolutionized this landscape by offering a container-based virtualization solution. This allowed developers to effortlessly bundle their applications into container images, complete with dependencies like libraries and customized configurations necessary for application execution. Docker's impact extends beyond simplifying development processes; it also offers significant cost benefits, making it a game-changer in the world of container technology. Docker's cost benefits can lead to significant savings, a prospect that should inspire optimism in any organization.

The whole IT industry changed because of Docker. An increasing number of organizations started to deploy their applications as Docker containers, which raised challenges regarding how to efficiently operate containers at a large scale. That is when technologies like Kubernetes appeared as a solution to managing containers. Before diving into Kubernetes, let us discover how Docker works.

Exploring Docker

It is relatively simple to operate a bunch of Docker containers without the assistance of any other tool than Docker itself. Containers run like processes in the operating system, so it is essential to ensure container processes are always running without errors. A system administrator can inspect container logs and restart containers when necessary if something goes wrong. For simple use cases, Docker alone may be enough to host applications. There may

be manual administration tasks to keep containers running, which is fine for smaller, non-critical systems. However, Docker may need to be complemented with other technologies to host critical enterprise applications.

Docker delivers effective container virtualization technology but lacks a reliable mechanism for operating containers in cluster-based environments that can meet strict application requirements involving high availability and fault tolerance.

With the increasing adoption of container technologies, container orchestrator solutions like Mesos, Marathon, Rancher, Docker Swarm, and Kubernetes appeared in the market. These container orchestrators were designed to facilitate the operation of critical applications running on containers. With a container orchestrator, we can adequately manage operating system resources like networks and storage to meet the requirements of container-based systems. Also, container orchestrators play a fundamental role in deploying the software by allowing deployment strategies that decrease the risk of application downtime.

We can only discuss Kubernetes by also discussing containers. Kubernetes exists only because of the container technology. So, to prepare ourselves for a deeper investigation of Kubernetes, we first explore Docker by examining its fundamentals.

Learning Docker fundamentals

Docker is supported on different operating systems, including Windows, MacOS, and Linux. To start playing with it, you can install either the Docker Desktop or the Docker Engine on your computer. The installation steps for Docker are outside the scope of this book, but you can find installation instructions at **https://docs.docker.com/manuals/**.

The Docker Engine provides the container virtualization engine and a CLI command tool for controlling Docker resources like images, containers, volumes, and networks. Docker Desktop includes the Docker Engine and a friendly graphical user interface that simplifies the management of Docker resources. It is an alternative for those who prefer the user interface over the CLI command tool.

To create a Docker container, we first need to create a Docker image using the Dockerfile:

```
FROM busybox
CMD ["date"]
```

The two lines above are placed in a file called Dockerfile, the default file name that Docker uses when building Docker images. There are two image types: parent-derived image and base image. Parent-derived images are always built on top of a parent Docker image. The example above uses the `FROM` directive to refer to the `busybox` parent image. Base images have no parent; they use the `FROM scratch` directive in their Dockerfile to indicate they do not refer to any parent image. We also use the `CMD` directive to execute a command once the container starts. A command is a program executed as an isolated process in the Docker container. The Dockerfile supports other directives, allowing us to customize Docker images by adding files, creating directories, creating environment variables, setting file system permissions, and other operations.

Managing Docker images

We can build a Docker image using the Dockerfile we created previously, using the following command:

```
$ docker build . -t busybox-date
```

The command above should be executed in your operating system terminal. The `docker` binary is the CLI tool provided by the Docker Engine. It allows several operations, such as building images with the `build` option. After the `build` command, the dot sign specifies the Dockerfile path location, which in our case, is the current directory represented by the dot sign. The `-t` option refers to the Docker image name, which in the command example is `buysbox-date`. When tagging a Docker image, we can specify its path and tag that can work as a versioning mechanism:

```
$ docker build . -t s4intlaurent/busybox-date:1.0
```

The image name provided by the `-t` option is based on the `namespace/repository:tag` structure. In the above example, `s4intlaurent` is the namespace from the Docker Hub, `busybox-date` is the repository, and `1.0` is the image tag. Docker Hub (**https://hub.docker.com/**) is a public image storage location, also known as a registry, where users can store their Docker images. Organizations may use a private registry with their own namespace.

If the version is omitted, the Docker image is automatically tagged as the latest.

You can list all Docker images stored in your computer using one of the following commands:

```
$ docker images
```

```
$ docker image ls
```

Or you can specify the Docker image tag you are looking for:

```
$ docker image ls s4intlaurent/busybox-date:1.0
```

```
REPOSITORY                              TAG        IMAGE
ID        CREATED          SIZE
```

```
s4intlaurent/busybox-date   1.0        ca915675332d
11 months ago   4.26MB
```

Once a Docker image tag is created locally, we can push it to a remote registry:

```
$ docker push s4intlaurent/busybox-date:1.0
```

We can pull a Docker image from a remote registry with the following command:

```
$ docker pull s4intlaurent/busybox-date:1.0
```

There can be multiple tags of the same Docker image repository, each having its value like `1.0`, `tag-a`, and `my-image-2.0`, for example.

Creating Docker containers

To create a container, we can refer to the Docker image repository name, followed by its tag, or the Docker image ID:

```
$ docker run s4intlaurent/busybox-date:1.0
```

```
Thu May  9 23:35:47 UTC 2024
```

By executing the `docker run`, we create a new container that executes the `date` command inside the container. After the executing `date`, the process is terminated, which means the container is also terminated. Remember, a container is attached to one or more processes. If no processes are executing, then the container is finished. To keep a container running, we need a daemon process. This kind of process is used in scenarios where a

process stays alive indefinitely and can receive and handle requests. We will explore this scenario later in this chapter when creating a Docker image of a Spring Boot application.

We can check what containers are currently running with the following command:

```
$ docker ps
```

If we pass the **-a** option, Docker will show all containers, including those that have been terminated:

```
$ docker ps -a
```

```
CONTAINER ID IMAGE                                COMMAND
CREATED         STATUS
```

```
32ff796facf8 s4intlaurent/busybox-date:1.0 "date"  15
minutes ago Exited(0)
```

```
15 minutes ago
```

The **STATUS** shows if the container is running or not. **Exited** means the container has been terminated.

The following is how we can kill a running container:

```
$ docker kill [CONTAINER ID]
```

Issuing a **docker kill** command immediately terminates the process inside the Docker container. After killing a container, we can altogether remove it from the system:

```
$ docker rm 32ff796facf8
```

```
32ff796facf8
```

The command above relies on the complete container ID. However, it is sometimes possible to pass only the first three characters of the ID string.

One of the practical benefits of removing a container is that it can free up storage space that the killed container might be utilizing, thereby optimizing your system's resources.

More advanced commands, like **docker volume**, let us create persistent storage that a container can use. By default, Docker container storage is ephemeral, so files created in a container will disappear once the container is terminated. The **docker network** command also allows us to create virtual network interfaces.

Let us explore now one of the most used container orchestrators, Kubernetes.

Introducing Kubernetes

Kubernetes started as an internal *Google* project called Borg, which used to operate containers on a large scale. *Google* eventually open-sourced the project and renamed it Kubernetes. It is considered the most used container orchestrator in the market. Kubernetes can operate containers in small devices like Raspberry PI or be used on top of an entire data center dedicated to containers.

Kubernetes has become popular, especially in large enterprises, because it provides a reliable platform for managing and running containers. Organizations found Kubernetes to be a mature technology capable of hosting mission-critical applications with high availability and fault tolerance.

The decision to use Kubernetes comes when operating containers solely with Docker, for example, is insufficient. Docker alone does not provide a high-availability solution for distributing container workloads across multiple cluster servers. Docker cannot dynamically auto-scale containers based on CPU or memory usage. Applying a load balancing mechanism on container network traffic is not possible with just Docker; however, it is possible when using Kubernetes.

So, Kubernetes gets the core container technology and surrounds it with additional components that make containers viable for running enterprise applications.

Most cloud providers offer Kubernetes-based solutions, and companies of all sizes rely on them to run their software. Designing an application to run on Kubernetes gives an organization the flexibility to change cloud providers without significant impact. Applications can be developed to rely on pure Kubernetes standards instead of features provided by a specific cloud provider. However, there are customized container orchestrators built on top of Kubernetes, like OpenShift from Red Hat and Kyma from SAP, that provide additional capabilities to ones already provided by Kubernetes.

A Java developer needs a basic understanding of how Kubernetes

works because a growing number of Java projects are running in Kubernetes clusters. So, next, we explore the fundamentals of Kubernetes architecture and some of its main objects commonly seen when deploying an application.

Kubernetes architecture

Kubernetes is a cluster made of at least one worker node, which is where containerized applications run. We can have Kubernetes running on a single or multiple machines. When running on a single machine, this machine acts simultaneously as the master and worker node. When running with multiple machines, master and worker nodes run in separate machines. Master nodes are responsible for cluster management activities, while worker nodes run containerized applications.

Inside a Kubernetes cluster, we have control plane and node components. Going next, we cover control plane components.

kube-scheduler

When Kubernetes objects need to be provisioned in one of the worker nodes, the kube-scheduler is responsible for finding a worker node that best suits the requirements of a given Kubernetes object. Worker nodes with high CPU, memory, and storage usage may be skipped in favor of worker nodes with more free capacity. A Pod is one of the Kubernetes object examples we will explore further in the next section.

kube-apiserver

The kube-apiserver interconnects different Kubernetes components and provides an external API that command-line tools like **`kubectl`** use to interact with the Kubernetes cluster. Practically everything that occurs inside a Kubernetes cluster passes through the kube-apiserver.

kube-controller-manager

Kubernetes objects have a current and desired state. The desired state is expressed through a **Yet Another Markup Language** (**YAML**) representation that describes how a Kubernetes object should be provided. If, for some reason, the current Kubernetes

object's current state is not the same as the desired one, then the **`kube-controller-manager`** may take action to ensure the desired state is achieved.

Control plane components usually run in a master node, a machine in the Kubernetes cluster used only for management purposes.

Next, we cover Kubernetes node components.

Container runtime

It is the container engine running in a Kubernetes node. It can be *containerd*, which Docker is based on, or any other compatible container technology. Kubernetes provides the **Container Runtime Interface** (**CRI**), which is a specification defining how container technologies can be implemented to be supported by Kubernetes, so any container runtime that implements such a specification is also compatible with Kubernetes.

kubelet

Every node in a Kubernetes cluster needs an agent called **kubelet** to manage containers running in the node machine. When adding a new worker node to an existing Kubernetes cluster, we need to ensure this node has the kubelet agent properly installed.

kube-proxy

Network access to containers managed by Kubernetes can be done through the network proxy provided by kube-proxy. We can use this proxy to set up a direct connection with one of the ports exposed by a container running in a Kubernetes cluster.

Node components run on every Kubernetes node, including master and worker nodes.

Next, we learn about Kubernetes objects and how they are used to run containerized applications.

Kubernetes objects

The containerized application represents the fundamental element by which all the Kubernetes machinery is driven. When hosting an application in Kubernetes, we can make it available for external networks, provide environment variables required by the

application, and control how many instances of the application will run simultaneously. Such tasks can be accomplished by using Kubernetes objects. We explore them further in the upcoming sections.

Pod

We do not deal directly with containers in a Kubernetes environment. Instead, we use Pods composed of one or more containers. A Pod acts like a wrapper that controls the entire lifecycle of a container. A **`Pod`** object specifies the container image that must be used when the Pod is deployed in a Kubernetes cluster. We have an example showing how to create a YAML representation of Pod, as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: httpd
spec:
  containers:
  - name: httpd
    image: httpd: 2.4
    ports:
    - containerPort: 80
```

When defining a Kubernetes object, we utilize a key component called **`kind`** that specifies the object's type, which in the example above is **`Pod`**. The **`metadata`** block contains elements that describe the Pod, including its **`name`**. Within the **`containers`** block, we can define one or more containers that are managed by this Pod. Each container is specified by its **`name`** and **`image`**. The **`ports`** block, specifically **`containerPort`**, allows us to define the port on which the containerized application is running.

Although possible, it is not a common practice to create Pods directly. Most of the time, Pods are managed by other Kubernetes objects like the **`Deployment`**. Let us check it next.

Deployment

Applications running on Kubernetes can scale horizontally, which means multiple instances of the same application can be provisioned to distribute application processing better and provide high availability. We can accomplish this by using the Deployment object that manages Pod objects. The Deployment lets us define, for example, how many replicas of a Pod must run simultaneously. Check the following example of the YAML representation of a Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd
  labels:
    app: httpd
spec:
  replicas: 3
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:2.4
        ports:
        - containerPort: 80
```

Part of the Deployment declaration is similar to a Pod because we need to define how Pods managed by the Deployment will be created. Note the **`replicas`** component; we use it to tell Kubernetes how many Pod instances must run under this Deployment. This YAML declaration expresses the desired state of a **`Deployment`** object. Suppose Kubernetes detects, for example, that only two Pod instances are running when three instances are the expected amount. In that case, Kubernetes automatically tries to bring up another instance to ensure the current state matches the desired state. Also note the **`matchLabels`** block; we use it to define labels other Kubernetes objects can use to refer to the Deployment object. A common use case for the **`matchLabels`** component is when we want to expose a Deployment in the network using a **`Service`** object. We explore **`Service`** objects next.

Service

Pods can communicate with each other and be externally accessible for clients outside the Kubernetes cluster. By default, when a Pod is deployed, it is not accessible by other Pods in the cluster. We can solve it by creating a **`Service`** object:

```
apiVersion: v1
kind: Service
metadata:
  name: httpd
  labels:
    app: httpd
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: httpd
```

The **selector** block may contain a reference pointing to the same label value used in a Deployment or Pod object. That is how a Service can expose other Kubernetes objects to the network. Note that we use **ClusterIP** as the Service type in the example above. We use this Service to expose a Pod to the internal Kubernetes cluster network, allowing Pods to communicate with each other through the Service. We can use the **NodePort** or **LoadBalance** Service type to expose a Pod to networks outside the Kubernetes cluster. A Service must specify its port and protocol. The **targetPort** component refers to the port the containerized application listens to in the Pod. When omitted, the **targetPort** is the same as the Service **port**.

ConfigMap and Secret

Containerized applications may depend on external data, such as environment variables and file configuration properties. Kubernetes provides the **ConfigMap** and **Secret** objects as mechanisms allowing the injecting of external data for applications running inside the Pods. The following is how we can create a ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-application
data:
  DATABASE_URL: "mysql://sample-database:3306/
test"
```

The example above provides an environment variable called **DATABASE_URL**. ConfigMap data can also be used to mount files inside a Pod.

A Secret is similar to ConfigMap, but it targets sensitive data. Following is a Secret example:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-application
```

```
type: Opaque
data:
  DATABASE_USERNAME: dGVzdAo=
  DATABASE_PASSWORD: cGFzcwo=
```

When providing data as environment variables in a Secret object, the values must be encoded with base64.

Understanding the most important Kubernetes objects is essential for developers preparing applications for Kubernetes deployment. We will explore next how to create a Docker image of a Java application based on the Spring Boot, Quarkus, and Jakarta EE frameworks.

Dockerizing a Spring Boot, Quarkus, and Jakarta EE application

Most Java frameworks provide mechanisms to generate a bootable JAR file containing all dependencies required to run the application from such a JAR file. Having a bootable JAR is essential for dockerizing Java applications. By dockerizing, we mean creating a Docker image using the bootable JAR file.

A bootable JAR file, also known as an uber of fat JAR, is a file that contains the compiled application's class files along with all the dependencies required to run the Java application. Such dependencies can also be included as compiled class files, resulting in a JAR file containing the application's compiled class files and their dependencies.

Next, we will explore how to prepare applications based on Spring Boot, Quarkus, and Jakarta EE to be executed inside a Docker container.

Creating a bootable JAR of a Spring Boot application

Spring Boot applications can rely on a Maven plugin, which creates a JAR file packed with all application classes and dependencies required to run them. Below is how the pom.xml file of a Spring Boot application can be configured to build a bootable JAR file:

```
<build>
```

```
    <finalName>sample-spring-boot-app</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</
groupId>
            <artifactId>spring-boot-maven-plugin</
artifactId>
            <version>3.3.4</version>
        </plugin>
    </plugins>
</build>
```

With the **`finalName`** property, we define the JAR file name. The plugin responsible for doing the magic is called **`spring-boot-maven-plugin`**. When using this plugin, the **`mvn package`** command will create the **`sample-spring-boot-app.jar`** inside the **`target`** directory of the Spring Boot Maven project.

Next, we will learn how to create a bootable JAR for a Quarkus application.

Creating a bootable JAR of a Quarkus application

The approach for creating a bootable JAR for Quarkus is similar to what we did for the Spring Boot application. The following example shows how we can configure the **`pom.xml`** file:

```
<build>
   <finalName>sample-quarkus-app</finalName>
    <plugins>
        <plugin>
            <groupId>${quarkus.platform.group-id}
</groupId>
            <artifactId>quarkus-maven-plugin</
artifactId>
            <version>${quarkus.platform.version}</
```

```
version>
            <extensions>true</extensions>
            <executions>
                <execution>
                    <goals>
                        <goal>build</goal>
                        <goal>generate-code</goal>
                        <goal>generate-code-
tests</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The example above relies on the variables **`quarkus.platform.group-id`** and **`quarkus.platform.version`** that you can define based on the Quarkus version you want to use. Quarkus provides a plugin called **`quarkus-maven-plugin`**, which gathers all application dependencies into a single JAR file. We must execute the **`mvn package`** command to create a JAR file called **`sample-quarkus-app.jar`** inside the **`target`** directory.

We use the **`build`** goal to package the Quarkus application. The **`generate-code`** goal compiles the source code files, whereas the **`generate-code-tests`** goal compiles test code files. Once both source and test code files are compiled, the **`build`** goal packages them into a bootable JAR file.

Following, we will learn how to create a bootable JAR for a Jakarta EE application.

Creating a bootable JAR of a Jakarta EE application

We need a certified Jakarta EE application server to run a Jakarta

EE application. When creating a bootable JAR, such a requirement can be fulfilled by embedding the application server into the same JAR file containing the Jakarta EE application compiled classes. Among the available certified Jakarta EE application servers, let us use the WildFly for configuring the **`pom.xml`**:

```
<build>
    <finalName>sample-jakartaee-app</finalName>
    <plugins>
        <plugin>
            <groupId>org.wildfly.plugins</groupId>
            <artifactId>wildfly-jar-maven-plugin</
artifactId>
            <version>11.0.2.Final</version>
            <configuration>
                <feature-pack-location>wildfly-
preview@maven(org.jboss.universe:community-
universe)</feature-pack-location>
                <layers>
                    <layer>jaxrs-server</layer>
                </layers>
                <plugin-options>
                    <jboss-fork-embedded>true</
jboss-fork-embedded>
                </plugin-options>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>package</goal>
                    </goals>
                </execution>
```

```
            </executions>
        </plugin>
    </plugins>
</build>
```

We rely on the **`wildfly-jar-maven-plugin`** to create a bootable JAR file containing a WildFly application server configured through the **`jaxrs-server`** property value to run Jakarta EE applications supporting the Jakarta RESTful Web Services specification. We can add a **`layer`** inside the **`layers`** block representing WildFly capabilities, which our Jakarta EE application requires. Executing the **`mvn package`** command will result in the creation of the **`sample-jakarta-app.jar`** file inside the **`target`** directory.

Having learned how to create bootable JAR files of well-known Java frameworks, let us see how we can use such JAR files to create a Docker image. The next section uses, as an example, the bootable JAR file of a Spring Boot application.

Creating the Docker image

The Dockerfile we use to create the Docker is usually placed in the Java Maven project's root directory. We can use the following code to create such a file:

```
FROM openjdk:21-slim
ENV JAR_FILE sample-spring-boot-app.jar
ENV JAR_HOME /usr/apps
COPY target/$JAR_FILE $JAR_HOME/
WORKDIR $JAR_HOME
ENTRYPOINT ["sh", "-c"]
CMD ["exec java -jar $JAR_FILE"]
EXPOSE 8080
```

We need the **Java Virtual Machine** (**JVM**) to run JAR files inside the container. That is why, in the example above, our image refers to a parent image called **`openjdk:21-slim`** that provides the JVM. Then, we use the **`ENV`** directive to set the **`JAR_FILE`** environment variable pointing to a JAR file called **`sample-spring-boot-`**

app.jar produced by Maven after the application is compiled and packaged. Another **ENV** directive sets the **JAR_HOME** environment variable to the directory where the JAR file will be placed in the container.

The **COPY** directive's first parameter, **target/$JAR_FILE**, refers to a relative path in the local machine building the Docker image. The second **COPY** directive parameter refers to the path in the container environment. The **COPY** directive copies files from the local machine to the container environment. The **WORKDIR** directive sets the container default directory from where commands are executed. Next, the **ENTRYPOINT** is used to run the container as an executable. When combined with the **CMD** directive, the **ENTRYPOINT** provides a command the container executes, and the **CMD** complements it by giving the command's parameters. The **ENTRYPOINT** starts a command shell that we use to instruct the JVM to run our application's JAR file. Finally, we use the **EXPOSE** directive to open the 8080 container port, the same port used by the Spring Boot application.

The following is how we can create the Docker image:

```
$ docker build . -t s4intlaurent/sample-spring-boot-
app:1.0
```

After creating the image, we can confirm if it is working by creating a container out of it:

```
$ docker run -d -p 8080:8080 s4intlaurent/sample-
spring-boot-app:1.0
```

The **-d** option instructs Docker to run the container in the background; the **-p 8080:8080** maps the host port 8080 to the container port, which is also 8080. By mapping a port to the container port, we make the container accessible to external users. We can check the container status by executing the following command:

```
$ docker ps

CONTAINER ID   IMAGE   COMMAND  CREATED
STATUS              PORTS


2dc5832882e1   s4intlaurent/sample-spring-boot-
app:1.0   "sh -c 'exec java -j…"   2 minutes ago   Up
```

```
2 minutes   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp
```

Notice the container **STATUS** is **Up**, which confirms the Spring Boot application is running inside the container.

We must upload the Docker image to the container registry, which we can use later when deploying the application to Kubernetes. That can be done by issuing a **docker push** command using the image tag we created previously:

```
$ docker push s4intlaurent/sample-spring-boot-app:1.0

The push refers to repository [docker.io/
s4intlaurent/sample-spring-boot-app]

5f70bf18a086: Layer already exists

8c17bbbc8b59: Pushed

659a8c4ba776: Pushed

0ac7ecf8a41c: Pushed

d310e774110a: Pushed

1.0: digest:
sha256:415a0b98a203b90548e2ef001fee9fd996f29c07d02a8df1dc30
size: 1371
```

The command above pushes the Docker image to the public Docker Hub registry accessible through **https://hub.docker.com/r/s4intlaurent/sample-spring-boot-app**.

Next, we learn how to deploy a Docker image into a Kubernetes cluster.

Deploying Docker-based applications on Kubernetes

To understand how Docker and Kubernetes can be used together, we will cover the steps to run a Spring Boot application in a Kubernets cluster. To prepare a Java system to run as a containerized application in a Kubernetes cluster correctly, we need to create a Docker image for it, push the image to a container registry, and provide the Kubernetes objects to deploy the Spring Boot application and make it available to clients outside the Kubernetes cluster. Instead of creating a new Spring Boot application from scratch, we use the one created previously in *Chapter 5*, Building production-grade systems with Spring Boot.

As we have already created the Docker image of our application in the previous section, let's learn how to externalize the application configuration through environment variables. Externalizing application configuration is common practice when containerizing application configuration.

Externalizing application configuration

Using environment variables allows you to change application configuration without recompiling it. That is also helpful when deploying the application in multiple environments where each environment may require specific settings like database host, user, and password. On Spring Boot, we can apply external configuration using placeholders when defining the **`application.yaml`** file:

```
spring:
  datasource:
    url: ${DATABASE_URL:jdbc:h2:mem:mydb}
    username: ${DATABASE_USERNAME:sa}
    password: ${DATABASE_PASSWORD:password}
    driverClassName:
${DATABASE_DRIVER:org.h2.Driver}
  jpa:
    database-platform:
${DATABASE_DIALECT:org.hibernate.dialect.H2Dialect}
```

For example, **`${DATABASE_URL:jdbc:h2:mem:mydb}`** defines the data source URL configuration. The **`DATABASE_URL`** is the environment variable expected to be provided by the environment where the application is running. It can be provided by our local machine or a container, for example. If the environment variable value is not defined, Spring Boot uses the fallback value as **`jdbc:h2:mem:mydb`**. Later in this chapter, we will define environment variable values using the **`ConfigMap`**.

Next, we learn how to create Kubernetes objects required to run the Java application in a Kubernetes cluster.

Creating Kubernetes objects

We can easily set up a Kubernetes cluster locally using a local cluster solution like minikube or kind. Installing a local Kubernetes cluster is out of the scope of this book, but you can find instructions to install it at **https://minikube.sigs.k8s.io/docs/start/**. Having a local Kubernetes cluster is very convenient for making sure the Kubernetes objects are working as expected.

After setting up a minikube cluster, we can create the Kubernetes objects to run our Spring Boot application. Let us start by creating a ConfigMap.

Providing application configuration with a ConfigMap

Remember, we used environment variables to externalize the properties used in the **`application.yaml`** file from Spring Boot. We need to create a file called **`configmap.yaml`** to provide the values for those environment variables:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample-spring-boot-app
data:
  DATABASE_URL: "mysql://sample-spring-boot-app-
mysql:3306/test"
  DATABASE_DRIVER: "com.mysql.cj.jdbc.Driver"
  DATABASE_DIALECT:
"org.hibernate.dialect.MySQL8Dialect"
```

The environment variables above are used to establish a connection with a MySQL database.

The following is how you can install the above ConfigMap in a Kubernetes cluster, given the object's file name is **`configmap.yml`**:

```
$ kubectl apply -f configmap.yaml
```

Note that there are no credentials data in the ConfigMap; we put this data into a Secret. Let us check it next.

Using a Secret to define database credentials

Like a ConfigMap, we can use a Secret to define environment variables containing base64 encoded values. There are many ways to encode a string; you can do it online using a base64 encoder website. Most operating systems like Windows, MacOS, and Linux also contain tools that let you encode strings. The following example shows how we can encode the database credentials using base64 on Linux:

```
$ echo test | base64
dGVzdAo=
$ echo pass | base64
cGFzcwo=
```

We first encode the **`test`** string, which is the database user. The second command encodes the **`pass`** string we use as the password to connect to the database. Following that, we use the base64 encoded values when creating the Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-spring-boot-app
type: Opaque
data:
  DATABASE_USERNAME: dGVzdAo=
  DATABASE_PASSWORD: cGFzcwo=
```

The content above is placed in a file called **`secret.yaml`**, which is used to create the **`Secret`** object in the Kubernetes cluster. Following is an example showing how we can install a **`Secret`** object:

```
$ kubectl apply -f secret.yaml
```

Having defined the **`ConfigMap`** and **`Secret`** objects containing database connection details for our Spring Boot application, we can create the **`Deployment`** objects to deploy the application.

Deploying the application with a Deployment

We want to use MySQL as the database for our Spring Boot application. That can be done by creating a **`database.yaml`** file configured to deploy a containerized MySQL server to the Kubernetes cluster:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-spring-boot-app-mysql
  labels:
    app: sample-spring-boot-app-mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample-spring-boot-app-mysql
  template:
    metadata:
      labels:
        app: sample-spring-boot-app-mysql
    spec:
      containers:
        - name: sample-spring-boot-app-mysql
          image: mysql:latest
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "pass"
            - name: MYSQL_DATABASE
              value: "test"
          ports:
```

```
            - containerPort: 3306
```

We explicitly define the **`MYSQL_ROOT_PASSWORD`** and **`MYSQL_DATABASE`** environment variable credentials required by the MySQL container image. Defining environment variables directly in a **`Deployment`** object is a more straightforward alternative than using a **`ConfigMap`** or **`Secrets`**.

Next, we define the **`Deployment`** object for our Spring Boot application by creating the **`deployment.yaml`** file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-spring-boot-app
  labels:
    app: sample-spring-boot-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample-spring-boot-app
  template:
    metadata:
      labels:
        app: sample-spring-boot-app
    spec:
      initContainers:
        - name: sample-spring-boot-app-mysql-init
          image: busybox
          command: [ 'sh', '-c', 'until nc -zv
sample-spring-boot-app-
mysql.default.svc.cluster.local 3306; do echo
waiting for sample-spring-boot-app-
```

```
mysql.default.svc.cluster.local; sleep 5; done;' ]
      containers:
        - name: sample-spring-boot-app
          image: s4intlaurent/sample-spring-boot-
app:1.0
          envFrom:
          - configMapRef:
              name: sample-spring-boot-app
          - secretRef:
              name: sample-spring-boot-app
          ports:
            - containerPort: 8080
```

Note that this Deployment has a **`initContainers`** block using a **`busybox`** image that executes a **Netcat** (**`nc`**) command to check if the MySQL database on the host **`sample-spring-boot-app-mysql.default.svc.cluster.local`** and port 3306 is accessible. We set the **`initContainers`** block here to prevent the situation where the Spring Boot application starts before the database, causing application startup errors. This technique ensures a Pod is only initialized after tasks from an **`init`** container are successfully executed. The hostname **`sample-spring-boot-app-mysql.default.svc.cluster.local`** refers to the Kubernetes service that the Spring Boot Application uses to connect to the database.

We are using the Docker image **`s4intlaurent/sample-spring-boot-app:1.0`**, which we pushed to the Docker Hub registry when creating the Docker image for our Spring Boot application. Kubernetes will pull this image when we create the application Pod.

Once the **`Deployment`** object is appropriately configured, we can install it with a command similar to the one shown below:

```
$ kubectl apply -f deployment.yaml
```

Let us finish the Kubernetes objects configuration by creating the required Service objects for the Spring Boot application.

Allowing access to the application with a Service

We need to create a Service object that allows the Spring Boot application Pod to connect to the MySQL server Pod and another Service that allows the Spring Boot application to be exposed to clients outside the Kubernetes cluster.

We define Service for the MySQL server Pod by creating a **`database-service.yaml`** file with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: sample-spring-boot-app-mysql
  labels:
    app: sample-spring-boot-app-mysql
spec:
  ports:
    - port: 3306
      protocol: TCP
  selector:
    app: sample-spring-boot-app-mysql
```

When we omit the **`Service`** type, it defaults to **`ClusterIP`**, which is used here because we want to expose the MySQL Server Pod only to the internal Kubernetes cluster network. The Service's name **`sample-spring-boot-app-mysql`** becomes part of the **fully qualified domain name** (**FQDN**) defined as **`sample-spring-boot-app-mysql.default.svc.cluster.local`**. The **`default`** term refers to the namespace where Kubernetes creates objects; **`svc`** stands for Service; and **`cluster.local`** refers to the Kubernetes cluster. Other Pods can access the Service using the simple hostname **`sample-spring-boot-app-mysql`** or the FQDN.

Next, we create a **`service.yaml`** file that defines a **`Service`** object that exposes the Spring Boot application to external clients:

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: sample-spring-boot-app
  labels:
    app: sample-spring-boot-app
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 30080
      protocol: TCP
  selector:
    app: sample-spring-boot-app
```

That is a **`NodePort`** Service we use to open a port in the Kubernetes cluster, letting external clients interact directly with the Spring Boot application Pod. We explicitly set the 30080 as the **`NodePort`**. When the **`NodePort`** is not defined, Kubernetes automatically assigns a port from the 30000-32767 range. The 8080 port value is used for internal cluster communication and is also used as a default value for the **`targetPort`** when that is omitted, which is the case in the example above. Remember, **`targetPort`** refers to the port where the Spring Boot application port listens.

The following is an example showing how we can install our **`Service`** object in a Kubernetes cluster:

```
$ kubectl apply -f service.yaml
```

Let us see how we can now provision all these Kubernetes object definitions we created into a Kubernetes cluster.

Using kubectl to install Kubernets objects

A common way to interact with a Kubernetes cluster is through a command-line tool called **`kubectl`**. You can find the installation and configuration instructions for **`kubectl`** at **https://kubernetes.io/docs/tasks/tools/#kubectl**.

When using **minikube**, it automatically configures **kubectl** to connect to your local Kubernetes cluster.

Assuming that the Kubernetes objects YAML files were created in a directory called **k8s**, the following is how we can install those objects in our Kubernetes local cluster:

```
$ kubectl apply -f k8s/

configmap/sample-spring-boot-app created

service/sample-spring-boot-app-mysql created

deployment.apps/sample-spring-boot-app-mysql created

deployment.apps/sample-spring-boot-app created

secret/sample-spring-boot-app created

service/sample-spring-boot-app created
```

This command creates on the Kubernetes cluster all the objects specified by YAML files inside the k8s directory.

You can check if the Kubernetes Pods were created by executing the following command:

```
$ kubectl get pods

NAME                                                READY
  STATUS    RESTARTS   AGE

sample-spring-boot-app-68bbfd8596-q5g5w             1/1
  Running   0          3m48s

sample-spring-boot-app-mysql-5bfdfbdf44-khtk9       1/1
  Running   0          3m48s
```

A similar command be executed to check the Service and other objects:

```
$ kubectl get service

NAME
TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)
       AGE

kubernetes
ClusterIP   10.96.0.1          <none>          443/TCP
        276d

sample-spring-boot-app                    NodePort
```

```
10.106.96.230 <none>        8080:30080/TCP    5m33s

sample-spring-boot-app-mysql      ClusterIP
10.110.167.11 <none>        3306/TCP          5m33s
```

The **kubectl get {objectType}** command lets us inspect the state of any Kubernetes object.

Let us wrap up now by compiling and running the sample project.

Compiling and running the sample project

The sample project is based on the containerized Spring Boot application we have worked with throughout this chapter.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2008**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine. You also need Docker and Minikube.

Execute the following command to compile the application:

```
$ mvn clean package
```

Maven will create a JAR file called **sample-spring-boot-app.jar**, which we use to create a Docker image using the following command:

```
$ docker build . -t {YOUR_DOCKER_HUB_ACCOUNT}/sample-spring-boot-app:1.0
```

Replace **{YOUR_DOCKER_HUB_ACCOUNT}** with your account from **https://hub.docker.com/**. You can create an account there for free if you do not have one yet.

After building the Docker image, push it to the public Docker registry:

```
$ docker push {YOUR_DOCKER_HUB_ACCOUNT}/sample-spring-boot-app:1.0
```

Pushing the image is necessary before installing the Deployment Kubernetes object.

Next, we need to adjust the file at **k8s/deployment.yaml** by replacing the Docker image **s4intlaurent/sample-spring-boot-app:1.0** to **{YOUR_DOCKER_HUB_ACCOUNT}/sample-**

`spring-boot-app:1.0` where `{YOUR_DOCKER_HUB_ACCOUNT}` is your Docker Hub account name.

Following that, we install Kubernetes objects in the Minikube cluster:

```
$ kubectl apply -f k8s/

configmap/sample-spring-boot-app created

service/sample-spring-boot-app-mysql created

deployment.apps/sample-spring-boot-app-mysql created

deployment.apps/sample-spring-boot-app created

secret/sample-spring-boot-app created

service/sample-spring-boot-app created
```

The following is how we can make requests to the Spring Boot application running on Kubernetes:

```
$ curl -X POST {MIKIKUBE_IP_ADDRESS}:30080/person -H
'Content-type:application/json' -d '{"email":
"john.doe@davivieira.dev", "name": "John Doe"}'

$ curl -s {MIKIKUBE_IP_ADDRESS}:30080/person/
john.doe@davivieira.dev
```

We can use the following command to get the Minikube IP address:

```
$ minikube ip
```

The IP returned allows access to the Pods running inside the Minikube cluster.

Conclusion

Cloud technologies like Docker and Kubernetes are familiar to most Java developers. Knowing how to use these technologies is fundamental for anyone interested in designing and operating Java cloud-native systems.

This chapter explored how crucial virtualization technology, especially container virtualization, is for any cloud-native environment. We learned that a container comprises one or more isolated processes provided by the namespace and cgroups Linux kernel's features. Known as one of the most popular container technologies, we discovered how Docker makes developers' lives

easier by providing a convenient way to package and deliver containerized applications through Docker images created with a Dockerfile. Going deeper into the containers, we learned how powerful Kubernetes is in providing a container orchestration solution that reliably hosts containerized applications. Finally, we applied techniques like configuration externalization on the Spring Boot application to make it cloud-native and ready to run in Kubernetes clusters.

In the next chapter, we will look at monitoring and observability, activities that play a fundamental role in the availability and reliability of Java applications running in production. We will learn how to implement distributed tracing with Spring Boot and OpenTelemetry. Also, we will explore handling logs using the **Elasticsearch**, **Fluentd, and Kibana** (**EFK**) stack.

Chapter 9
Learning Monitoring and Observability Fundamentals

Introduction

Those tasked with supporting Java applications in production understand the value of comprehending system behavior in various situations. This understanding is built on the foundation of monitoring and observability techniques, which leverage metrics, logs, events, and other data to predict or address application failures. By looking at the basics of monitoring and observability, we can make informed decisions about the most effective technologies and approaches to swiftly respond to unexpected application behaviors.

Structure

The chapter covers the following topics:

- Understanding monitoring and observability

- Implementing distributed tracing with Spring Boot and OpenTelemetry
- Handling logs with Elasticsearch, Fluentd, and Kibana
- Compiling and running the sample project

Objectives

By the end of this chapter, you will learn the main concepts behind monitoring and observability. With these concepts as a solid foundation, you will learn how to apply distributed tracing techniques to understand through traces the life cycle of requests spanning multiple microservices. Finally, you will know how to collect and see logs from a Java application using Elasticsearch, Fluentd, and Kibana.

Understanding monitoring and observability

The crucial moment for software developers is when their applications go live. For backend developers, in particular, it is when their software is deployed to production environments. Although what backend developers deploy most of the time are not user-facing features, they do deploy backend software components that may support such features provided by the frontend system. It is fundamental to remember that the well-functioning of these backend components directly impacts the user-facing features, underscoring the importance of the backend development in ensuring a seamless user experience.

Imagine you are working as a backend developer. In that case, chances are high that you will face a scenario similar to the one described above because most backend development nowadays is based on server-side applications often integrated with frontend applications. Depending on the organization's structure, developers will be in charge of ensuring their applications run well in production. It can also happen that this responsibility will instead fall on the shoulders of application support analysts or system administrators. Either way, someone needs to be capable of understanding application behaviors, predicting and preventing issues, and identifying and fixing those issues when they occur.

For many years, developers, system administrators, support analysts, and anyone interested have been using monitoring tools and techniques to gain visibility and understanding of how server-side applications behave when serving requests and processing data. However, with the rise of distributed architecture applications, traditional monitoring approaches were found to have limitations. This led to the emergence of observability, a more comprehensive approach that provides deeper insights, especially in distributed architectures like microservices.

Before we explore observability, let us check next what is monitoring.

Monitoring

Any application behaves in its own way, given the constraints and load exerted upon it. The constraints are the computing resources, like CPU, memory, storage, and network bandwidth, available to the application to perform its activities. The load refers to how much of the available computing resources the application uses to carry on with its tasks. Routinely inspecting how an application behaves in the face of its constraints and load is one form of monitoring. The goal of this kind of monitoring is, for example, to prevent resource bottlenecks like lack of storage space or memory. That may be accomplished by setting up monitoring dashboards and alerts configured to send messages or phone calls when a predefined threshold is met, such as 80% of the disk being used.

Another form of monitoring is concerned with application logic. Some applications are developed in an enterprise environment to solve business problems. The logic to solve those problems may be susceptible to dependencies like database availability, input data provided through an application request, or data obtained from an API. The ability to inspect how well an application solves business problems is fundamental to predicting or quickly remediating issues. Monitoring application logic can be accomplished, for example, through the usage of metrics and application logs. Dashboards and alerts can also be used on top of data provided by metrics and logs.

Observability techniques were conceived to enhance standard monitoring practices and achieve a holistic understanding of what

happens in a software system by considering the behavior of not only a single application but also the relationship between multiple of them, as in the case of distributed architecture systems like microservices. Let us explore more of it next.

Observability

For quite some time, the standard way to build server-side systems was by developing a single monolith application. The focal point for all monitoring activities would be around that single monolith application, resulting in the creation of monitoring dashboards based on application metrics. These dashboards, a cornerstone of our understanding of the application's behavior, were heavily relied upon by developers and other interested parties. They served as a window into the application's world, providing crucial insights and supporting troubleshooting activities. The information obtained from the monitoring dashboards could trigger further investigation of application logs to identify an issue's root cause.

A problem arose when server-side systems started to be developed based on distributed architectures. The logic from a distributed system is scattered across multiple applications having particular responsibilities. Instead of having a single monolith application, several smaller applications are now working together to provide system functionalities. The shift in how server-side software is developed, from monolith to distributed, also triggered a change in techniques to understand how distributed software behaves, which ultimately culminated in what is called observability.

Observability is the ability to understand software system behaviors through structured events containing contextual data. It aims to enable the discovery of what, when, where, and why something happened in a system. Such contextual data is made of high-cardinality and high-dimensionality data.

A structured event is a piece of data describing system behavior at a given time. Its attributes are arbitrarily defined to provide as much context as possible for what happened when the software system attempted to do something.

High cardinality refers to data uniqueness. An example of high-cardinality data is an event having attributes like the Request ID that must store unique values in a system. On the other hand, a low-

cardinality data example would be an event having the Country attribute, which can contain non-unique values. High-cardinality data is one of the cornerstones of observability because it allows us to identify events describing system behaviors accurately.

High dimensionality refers to the data attributes used in an event describing system behavior. An event containing a comprehensive set of attributes is helpful in understanding system behaviors from different dimensions. For example, User ID, Organization ID, Region, Status, Source, or Destination can be used as dimensions where User ID is one dimension, Organization ID is another, and so on. An event lacking crucial attributes may compromise the comprehension of system behaviors; that is why it is essential to ensure structured events have high dimensionality based on relevant data attributes.

Structured events with high-cardinality and high-dimensionality data are the foundation for observability tools and techniques. In distributed systems, observability techniques can be used to understand the flow of a request going through multiple applications. Next, we learn how to implement distributed tracing using Spring Boot and OpenTelemetry.

Implementing distributed tracing with Spring Boot and OpenTelemetry

Understanding system behavior is a non-trivial challenge in distributed architecture systems. In such an architecture, a user request may trigger other requests on different applications that work together to provide a system functionality. For example, a user sends a request to Service A, which sends a request to Service B, which sends another request to Service C. If something goes wrong in one of the three services involved in the operation, we must be able to identify in which service the problem is coming from. We can do that using distributed tracing.

Distributed tracing is the technique that helps us understand system behavior based on traces emitted by an application. A trace represents the path a request takes to execute a system behavior. Every trace comprises one or more spans, representing a unit of work. As the request goes from one application to another, new spans are generated, carrying contextual data that lets us know the

previous span and to which trace all the spans are associated. With traces and spans, we can better see what happens in a system composed of multiple applications. We have an illustration of what a trace looks like as follows:

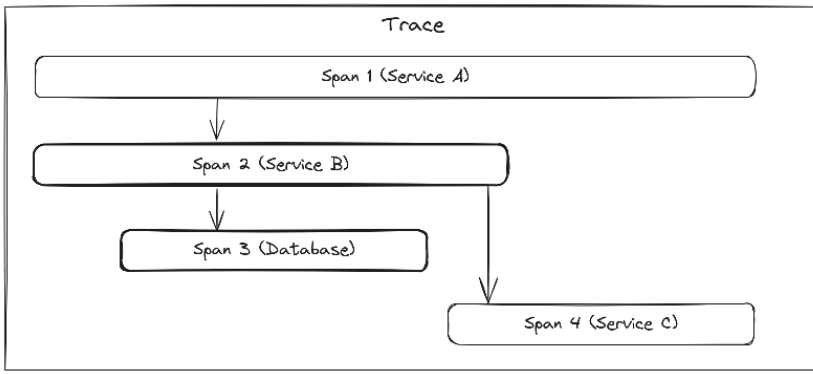

Figure 9.1: *The Trace structure*

The trace example starts with Span 1, which represents a request coming to Service A. Span 2 shows us that Service A requested Service B, which, in turn, made a request to the database, represented through Span 3. Finally, we can see that Service B requested Service C after receiving a response from the database.

OpenTelemetry provides a set of SDKs, APIs, and libraries that let us instrumentalize applications to generate telemetry data such as metrics, logs, and traces like the one presented in the previous example. Once adopted, OpenTelemetry also enables us to collect and export traces to observability applications, like Jaeager, that let us visualize the traces generated by a system. Having a way to visualize system traces is very helpful for troubleshooting purposes.

Let us start by building a simple distributed system based on two Spring Boot applications. That system will serve as the scenario for implementing distributed tracing using OpenTelemetry.

Building a simple distributed system

The system we will create is responsible for generating reports based on data stored in an inventory. Following a distributed architecture approach, we build a service responsible for generating reports and another service responsible for providing inventory

data.

Let us start by defining Maven's **`pom.xml`** with the required dependencies for both services.

Configuring dependencies

We start by defining the base Maven's project structure in the **`pom.xml`** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
         xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</
groupId>
        <artifactId>spring-boot-starter-parent</
artifactId>
        <version>2.7.18</version>
        <relativePath/>
    </parent>

    <groupId>dev.davivieira</groupId>
    <artifactId>chapter09</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <properties>
```

```
        <maven.compiler.source>21</
maven.compiler.source>
        <maven.compiler.target>21</
maven.compiler.target>
    </properties>

    <modules>
        <module>inventory-service</module>
        <module>report-service</module>
    </modules>
<!-- Code omitted -->
</project>
```

Note that this is a Maven multi-module project with a module for the inventory service and another for the report service. All the dependencies and build configurations defined in this pom.xml file are shared with the inventory service and report service modules, which will be defined soon.

Spring Boot simplifies our task by providing support for OpenTelemetry libraries. This allows us to enable distributed tracing effortlessly. Here is how we configure the first part of Maven's dependencies for Spring Boot and OpenTelemetry:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</
groupId>
            <artifactId>spring-cloud-
dependencies</artifactId>
            <version>2021.0.5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

```
        <dependency>
            <groupId>org.springframework.cloud</
groupId>
            <artifactId>spring-cloud-sleuth-otel-
dependencies</artifactId>
            <version>1.1.2</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

We use the **`dependencyManagement`** block to get the spring-cloud-dependencies and **`spring-cloud-sleuth-otel-dependencies`** POM dependencies. From the POM dependencies, we can specify the JAR dependencies, which we will do next:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
        <artifactId>spring-boot-starter-web</
artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</
groupId>
        <artifactId>spring-cloud-starter-sleuth</
artifactId>
        <exclusions>
            <exclusion>

<groupId>org.springframework.cloud</groupId>
```

```
                <artifactId>spring-cloud-sleuth-
brave</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</
groupId>
        <artifactId>spring-cloud-sleuth-otel-
autoconfigure</artifactId>
    </dependency>
    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-exporter-otlp</
artifactId>
        <version>1.23.1</version>
    </dependency>
</dependencies>
```

We use the **`spring-boot-starter-web`** dependency to create REST API endpoints for our services. The **`spring-cloud-starter-sleuth`** dependency provides auto-configuration for distributed tracing on Spring Boot applications. We exclude the **`spring-cloud-sleuth-brave`** because it is a trace generator library. Instead, we use the **`spring-cloud-sleuth-otel-autoconfigure`** dependency, which generates traces using OpenTelemetry. The **`opentelemetry-exporter-otlp`** lets us collect trace data and export it to a collector.

Let us now start implementing the distributed system with the inventory service.

Implementing the inventory service

The inventory service exposes a single REST endpoint that can be used by the report service to get inventory data:

```
@RestController
@RequestMapping("/inventory")
public class InventoryEndpoint {

    private static final Logger LOGGER =

LoggerFactory.getLogger(InventoryEndpoint.class);

    @GetMapping
    public List<String> getAllInventory() {
        LOGGER.info("Getting all inventory
items");
        return List.of("Inventory Item 1",
"Inventory Item 2", "Inventory
        Item 3");
    }
}
```

We use the **`@GetMapping`** annotation to expose inventory data through the GET endpoint at /inventory. That is all we need from the Java class implementation perspective. Now, we need to enable the application to produce traces. How we can do that using the application.yml file is shown as follows:

```
server:
  port : 8080

spring:
  application:
    name: inventory-service
  sleuth:
    otel:
      config:
```

```
      trace-id-ratio-based: 1.0
    exporter:
      otlp:
        endpoint: http://collector:4317
```

There are three essential configurations to pay attention to here:

1. The **`name`** property is used to group trace data based on the application's name.
2. The **`trace-id-ratio-based`** defines the ratio through which spans will be captured. The number **`1.0`** means all spans will be captured.
3. The **`endpoint`** property sets the collector's URL where trace data will be exported.

The trace collector is an external system that must be available. Otherwise, our Spring Boot application won't be able to export trace data. We will see soon how to provide such a collector system.

Using Docker Compose, we intend to provide the inventory service as a containerized application. To do so, we need to create a Dockerfile:

```
FROM openjdk:21-slim
ENV JAR_FILE inventory-service-1.0-SNAPSHOT.jar
ENV JAR_HOME /usr/apps
COPY target/$JAR_FILE $JAR_HOME/
WORKDIR $JAR_HOME
ENTRYPOINT ["sh", "-c"]
CMD ["exec java -jar $JAR_FILE"]
EXPOSE 8080
```

Note the port 8080 we expose in the Dockerfile is the same port used in the application.yml file from the Spring Boot application.

Let us now implement the report service

Implementing the report service

The report service communicates directly with the inventory

service. Let us implement a Java class that accomplishes that:

```
@RestController
@RequestMapping("/report")
public class ReportEndpoint {

    private static final Logger LOGGER =
    LoggerFactory.getLogger(ReportEndpoint.class);
    private final RestTemplate restTemplate;
    @Value("${inventoryService.baseUrl}")
    private String baseUrl;

    @Autowired
    public ReportEndpoint(RestTemplate
restTemplate) {
        this.restTemplate = restTemplate;
    }

    @GetMapping(path = "/generate")
    public List<String> generateReport() {
        LOGGER.info("Generating report");
        return getInventoryItems();
    }

    private List<String> getInventoryItems() {
        LOGGER.info("Getting inventory items");
        ResponseEntity<String[]> response =
        restTemplate.getForEntity(baseUrl + "/
inventory",
        String[].class);
```

```
        return
List.of(Objects.requireNonNull(response.getBody()));
    }
}
```

We use the **`RestTemplate`**, which provides a client that lets us communicate with other applications using the HTTP protocol. The **`generateReport`** method is called when the application receives a GET request at **`/report/generate`**. The **`generateReport`** method calls **`getInventoryItems`**, which contains the logic responsible for sending a GET request to the **`/inventory`** endpoint from the inventory service application. The implementation is simple but enough to show us how distributed tracing works.

Next, we configure the application.yml file:

```
server:
  port : 9090

spring:
  application:
    name: report-service
  sleuth:
    otel:
      config:
        trace-id-ratio-based: 1.0
      exporter:
        otlp:
          endpoint: http://collector:4317

inventoryService:
  baseUrl: ${INVENTORY_BASE_URL:http://
localhost:8080}
```

Note that the configuration is quite similar to the inventory service.

The only differences are the server port, which is 9090, the application's name, **`report-service`**, and the presence of the **`baseUrl`** property containing the inventory service URL.

The Dockerfile configuration is also quite similar to the inventory service:

```
FROM openjdk:21-slim

ENV JAR_FILE report-service-1.0-SNAPSHOT.jar

ENV JAR_HOME /usr/apps

COPY target/$JAR_FILE $JAR_HOME/

WORKDIR $JAR_HOME

ENTRYPOINT ["sh", "-c"]

CMD ["exec java -jar $JAR_FILE"]

EXPOSE 9090
```

To ensure the Spring Boot application will be accessible externally, we expose the 9090 port, the same port configured in the Spring Boot application.

At this stage, we have two Spring Boot applications: the inventory and report services. To complete the setup, we need to configure Docker Compose. This configuration not only starts both applications but also provides an instance of the Jaeger and OpenTelemetry Collector. These tools are crucial as they enable us to get and visualize application traces.

Setting up Docker Compose, Jaeger, and Collector

Using Docker Compose, we can bring up the inventory and report services and the distributed tracing tools Jaeger and OpenTelemetry Collector. Jaeger exposes tracing data to a user interface, showing application traces. The OpenTelemetry Collector provides tracing data collected from traces generated by the inventory and report services. The following code is how we can configure the docker-compose.yml file:

```
version: '3.8'

services:

  inventory-service:
```

```
    build: inventory-service/
    ports:
      - "8080"
  report-service:
    environment:
      - INVENTORY_BASE_URL=http://inventory-
service:8080
    build: report-service/
    ports:
      - "9090:9090"
  jaeger-service:
    image: jaegertracing/all-in-one:latest
    ports:
      - "16686:16686"
      - "14250"
  collector:
    image: otel/opentelemetry-collector:0.72.0
    command: [ "--config=/etc/otel-collector-
config.yml" ]
    volumes:
      - ./otel-collector-config.yml:/etc/otel-
collector-config.yml
    ports:
      - "4317:4317"
    depends_on:
      - jaeger-service
```

Observe that in the collector's configuration, we mount the file otel-collector-config.yml as a container volume file inside the container. Following is how the otel-collector-config.yml should look like:

```
# Code omitted
```

```
exporters:
  logging:
    loglevel: debug
  jaeger:
    endpoint: jaeger-service:14250
    tls:
      insecure: true

service:
  pipelines:
    traces:
      receivers: [ otlp ]
      processors: [ batch ]
      exporters: [ logging, jaeger ]
```

Note that we have specified **`jaeger-service:14250`** as the Jaeger endpoint. The host **`jaeger-service`** and **`14250`** port are part of the Docker Compose configuration defined previously.

With Jaeger and OpenTelemetry Collector integrated with traces produced by our Spring Boot applications, we can see how distributed tracing works in practice. Before we start playing with it, let us add an essential element to our observability setup: centralized logging.

Handling logs with Elasticsearch, Fluentd, and Kibana

We can identify what is happening and where something went wrong with metrics and traces, but we may need more information to tell us why something went wrong. Sometimes, the answers to the root cause of a problem can only be found after inspecting application logs. Anyone with experience troubleshooting applications knows that. The problem symptom usually starts, in the best-case scenarios, as an alert saying the system is not behaving as expected. In the worst-case scenarios, it starts with customers

complaining they cannot use the system. Either way, we must collect evidence to understand why the issue is happening and apply a fix as soon as possible.

Logs are fundamental in monitoring and observability because they contain information about application behavior. For quite some time, developers and system administrators dug directly into the server for application logs. That approach works great for monolith systems running on a single server. However, when you have distributed systems composed of many applications running as multiple instances across multiple servers to ensure high availability, the simple approach of digging directly into the server logs is no longer feasible. In such scenarios, the vast amount of logs from such diverse sources requires a centralized approach capable of aggregating logs and allowing people to navigate them easily.

Logs from different sources can be centralized in one place using technologies like Fluentd, Elasticsearch, and Kibana. Let us examine the purpose of each technology.

Fluentd

Known as data collection software, Fluentd is frequently used to capture application log outputs and send them to search engines like Elasticsearch. It has been widely adopted in distributed architecture systems running in a Kubernetes cluster.

Elasticsearch

Based on the Lucene search engine library, Elasticsearch is a distributed enterprise search engine often used with data collection systems like Logstash and Fluentd. Elasticsearch stores data in indexes and allows for the fast search of large amounts of data.

Kibana

Built specifically to work with Elasticsearch, Kibana is a system that provides a user interface for managing and searching Elasticsearch data. Users can search data using customized filters and criteria, allowing high flexibility in the search.

The three technologies, **Elasticsearch, Fluentd, and Kibana**, are commonly used together, forming the reliable and widely used **EFK**

stack. In the next section, we will guide you through setting up an EFK stack to capture logs from the Spring Boot applications we developed earlier.

Setting up EFK stack with Docker Composer

There is no need to change the application code because Fluentd captures data using the log output produced by the application. So, ensuring proper communication between applications generating logs and the Fluentd server is essential. We also need to ensure Fluentd sends log data to the Elasticsearch server that Kibana uses to display logs through a user interface.

1. Let us start by configuring the EFK stack in the docker-compose.yml file. The code is as follows:

```
version: '3.8'

services:

  fluentd:
    build: ./fluentd
    volumes:
      - ./fluentd/conf/fluent.conf:/fluentd/etc/flue
    links:
      - "elasticsearch"
    ports:
      - "24224:24224"
      - "24224:24224/udp"

  elasticsearch:
    image: elasticsearch:8.13.4
    container_name: elasticsearch
    environment:
```

```
      - discovery.type=single-node
      - xpack.security.enabled=false
    expose:
      - "9200"
    ports:
      - "9200:9200"
  kibana:
    image: kibana:8.13.4
    environment:
      - XPACK_SECURITY_ENABLED=false
      - ELASTICSEARCH_HOSTS=http://elasticsearch:920
      - INTERACTIVESETUP_ENABLED=false
    links:
      - "elasticsearch"
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
```

Elasticsearch and Kibana require security mechanisms by default. For the sake of simplicity, we are turning them off through environment variables like **`XPACK_SECURITY_ENABLED=false`** and **`INTERACTIVESETUP_ENABLED=false`**.

2. For the Fluentd setup, we need to provide a customized Fluentd Docker image configured with the Elasticsearch plugin. The following code shows how the Dockefile for such an image should be defined:

```
FROM fluent/fluentd:v1.17
```

```
USER root

RUN ["gem", "install", "fluent-plugin-elasticsearch"

USER fluent
```

Note we also, the docker-compose.yml refers to the file **`./fluentd/conf/fluent.conf`**, which specifies configurations like the IP address and port the Fluentd will listen to receive application logs, and also the destination place, Elasticsearch in our case, where log data will be sent:

```
<source>

  @type forward

  port 24224

  bind 0.0.0.0

</source>


<match *.**>

  @type copy

  <store>

    @type elasticsearch

    host elasticsearch

    port 9200

    logstash_format true

    logstash_prefix fluentd

    logstash_dateformat %Y%m%d

    include_tag_key true

    type_name access_log

    tag_key @log_name

    flush_interval 1s
```

```
  </store>

  <store>
    @type stdout
  </store>
</match>
```

The source configuration block defines a TCP endpoint through the **`@type forward`** option that accepts TCP packets from applications sharing their output. Note port 24224's definition; we use it when configuring the Spring Boot application container's logging on Docker Compose. After the **`source`** block, we have a **`match`** block to determine the log output destination. Two necessary configurations here are the **`host`** and **`port`**, which are defined as **`elasticsearch`** and **`9200`**, respectively.

3. Next, we add to the **`docker-compose.yml`** the configuration for the inventory and report service Spring Boot applications we built in the previous section:

```
# Code omitted
inventory-service:
  build: inventory-service/
  ports:
    - "8080"
  links:
    - fluentd
  logging:
    driver: "fluentd"
    options:
      fluentd-address: localhost:24224
      tag: inventory.service
```

```
report-service:
  environment:
    - INVENTORY_BASE_URL=http://inventory-service:80
  build: report-service/
  ports:
    - "9090:9090"
  links:
    - fluentd
  logging:
    driver: "fluentd"
    options:
      fluentd-address: localhost:24224
      tag: report.service
```

Note that inventory and report services have a `logging` configuration block that defines Fluentd as the log driver connecting to the **`localhost:24224`**. Remember, we previously configured Fluentd to listen in port 24224. What happens here is that logs produced by inventory-service and report-service containers will be forwarded to the Fluentd server.

Let us combine all the pieces and play with our observability engine, which supports distributed tracing and centralized logging.

Compiling and running the sample project

The sample project is based on the two Spring Boot applications developed throughout this chapter and the observability setup based on OpenTelemetry, Jaeger, and the EFK stack.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2009**.

You need to install the JDK 21 or above and Maven 3.8.5 or above on your machine. You also need Docker and Docker Compose.

Execute the following command to compile the two Spring Boot applications:

```
$ mvn clean package
```

Maven will create a JAR file for the inventor-service and report-service applications. Next, you can execute the following command to bring up all the containers from the Spring Boot applications and also the observability tools:

```
$ docker-compose up -d --build
```

After having all containers up and running, you use the following command to send a sample request to the report service:

```
$ curl -s localhost:9090/report/generate | jq
[
  "Inventory Item 1",
  "Inventory Item 2",
  "Inventory Item 3"
]
```

You can visualize application traces by accessing the Jaeger UI URL at **http://localhost:16686**. The screen is as follows:

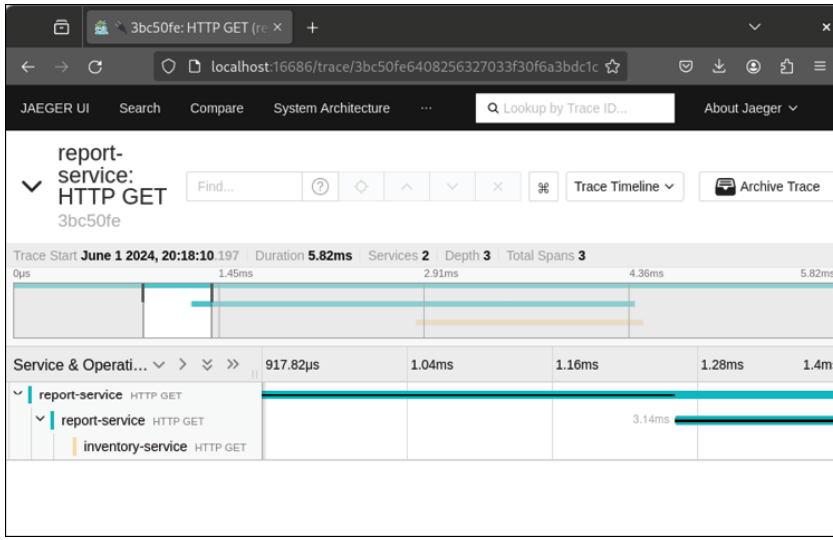

Figure 9.2: *The Jaeger UI*

Note that the trace comprises spans showing requests crossing through the report-service and inventory-service applications.

We can find application logs on Kibana at the URL **http://localhost:5601**. The screen is as follows:

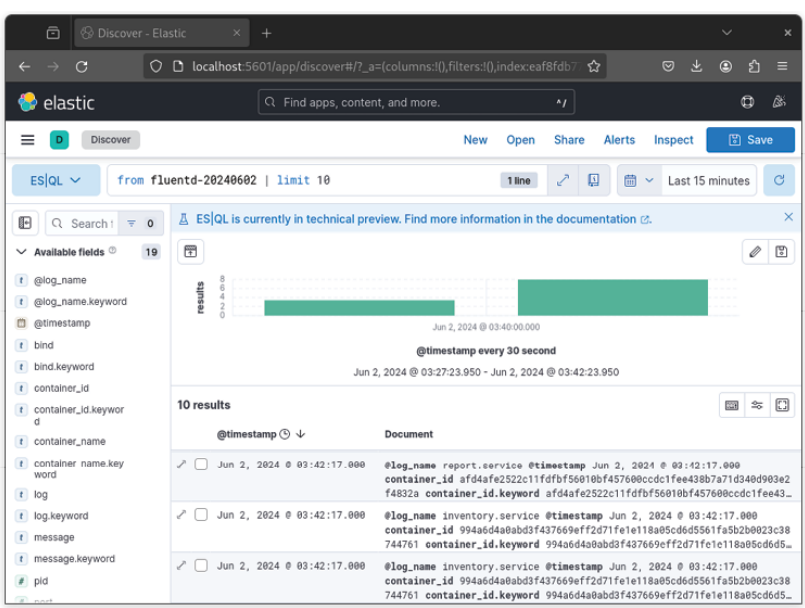

Figure 9.3: Kibana showing application logs

When accessing Kibana for the first time, click on the **Explore on my own** option, then on **Discover**, and finally, **Try ES|QL**, which lets you see data without configuring indexes.

Conclusion

In this chapter, we explored monitoring and observability. We learned that monitoring refers to the traditional approach of understanding system behavior through metrics, logs, dashboards, and alerts. We discovered that observability complements traditional monitoring techniques and is especially helpful for understanding the behavior of systems based on distributed architectures, like microservices.

We implemented a simple distributed system based on two Spring Boot applications that can generate traces. On top of those traces, we configured OpenTelemetry Collector to capture application traces and send them to the Jaeager, which lets us visualize traces and their spans.

Finally, we configured centralized logging using the **Elasticsearch, Fluentd, and Kibana** (**EFK**) stack, a solution that aggregates logs from different applications.

In the upcoming chapter, we will look at the exciting world of *Micrometer*, a library that has the potential to enhance the observability of Java applications significantly. We will learn the importance of providing application-specific metrics to enable better monitoring of application behaviors and explore how to implement such metrics.

Chapter 10

Implementing Application Metrics with Micrometer

Introduction

Understanding how an application behaves through metrics can help us remediate issues faster or prevent problems from getting worse. Metrics play a fundamental role in monitoring because they let us capture the state of the functionalities provided by the application. When interpreted, those states can indicate whether things are working as expected or if there are deviations requiring further investigation.

To harness the power of metrics in Java applications, we need to adjust them by configuring metrics to track application behaviors. We can do this using Micrometer, a well-known Java library that allows us to instrumentalize Java applications to generate metrics. That is why, in this chapter, we will explore how the Micrometer works, the metrics types it provides, and how and when we should use these metric types.

Structure

The chapter covers the following topics:

- Providing application metrics
- Introducing Micrometer
- Using Micrometer and Spring Boot to implement metrics
- Compiling and running the sample project

Objectives

By the end of this chapter, you will learn the benefits of instrumentalizing Java applications to generate metrics that describe how well the system is performing its activities. Knowing the advantages of application-specific metrics, you will also learn how to implement them using Micrometer, a powerful metrics library for Java.

Providing application metrics

Software systems are made of a set of instructions aimed at solving real-world problems. As programmers, we must understand which kind of real-world problems our applications intend to solve. It does not mean extensive expertise in a given field, like finance or medicine, is required to develop functional software. We do not need to be bankers to create good banking software. However, having substantial problem domain knowledge is invaluable for any developer. Such knowledge usually frames a specific aspect of an area that, for some reason, needs software to solve specific problems. So, the developed application should contain solutions for such specific problems.

The solutions software provides are materialized through a set of application behaviors that handle data by rules and instructions defined by business logic code. The business logic in enterprise systems is the codified expression of real-world problems. The enterprise system term is being emphasized to reinforce the focus on applications developed in a commercial, quite often corporate, context. As the codified representation of real-world problems, the business logic code plays a critical role in the success of any

enterprise application. Knowing that the business logic code is vital, how can we monitor it so that it works as expected once the application is up and running in production? Let us check it next.

There are two significant journeys in the software development lifecycle. The first journey is where problem-solving ideas are turned into working code. The second journey starts when the software is deployed to production and made available to users. Only after users begin to use our software, quite often in ways we cannot predict, can we really learn how well the business logic is fulfilling users' expectations. Such learning can occur the hard way when something goes wrong, and we discover it because a user reported application failure. We can also learn the effectiveness of business logic code through application-specific metrics implemented in strategic system locations to track critical application behaviors. Having application-specific metrics placed in the right system locations gives us the necessary visibility to predict potential failures and the ability to take action before the problem becomes too big and causes a significant impact on the users. There will be cases in which prediction will not be possible. Still, the troubleshooting time will be far shorter if the metrics are used as a starting point to investigate the issue's root cause and potential solutions.

It is important to remember that employing application-specific metrics is not a one-time activity. We usually start with metrics based on initial assumptions regarding what behaviors should be tracked in the application; then, we add more metrics based on what we learn by watching how the application reacts when users use it.

Having covered why we should implement metrics in our applications, let us learn about the Micrometer library, which helps us instrument Java applications with metrics.

Introducing Micrometer

Designed to be an agnostic metrics solution, Micrometer is a powerful library that lets us instrumentalize Java systems to produce metrics. It is considered agnostic because the Micrometer metrics are not vendor-specific; they can work with different monitoring vendors. Micrometer metrics are compatible, for

example, with tools like *Elastic* and *Dynatrace*, to name a few. So, using Micrometer allows us to switch across monitoring vendors without having to refactor the Java application.

Micrometer works so that the metrics it produces are shared by the instrumentalized Java application that exposes an endpoint used by external monitoring tools like *Prometheus*, for example, that record metrics in a time series database. Having a place to record the metrics produced by a Java application is beneficial because once the application is restarted, the metrics collected before are gone unless we have stored them elsewhere.

This section explores some core Micrometer concepts, like the registry and meters, and some of the most used meter types, including counters, gauges, timers, and distribution summaries. Let us proceed by checking what is a registry in Micrometer.

Registry

The registry is represented through the **`MeterRegistry`** interface, which acts as the fundamental component of the Micrometer architecture. All metrics produced by the Micrometer come from a registry that enables the creation of different metric types, like counters and gauges.

The **`MeterRegistry`** is an interface with implementations supporting multiple monitoring systems. The **`SimpleMeterRegistry`** class, for example, can be used for testing purposes because it keeps metrics data only in memory. For real-world scenarios, you might use the **`PrometheusMeterRegistry`** if your monitoring tool is Prometheus. You can create a **`SimpleMeterRegistry`** using the following code:

```
MeterRegistry registry = new
SimpleMeterRegistry();
```

By default, a registry publishes metrics only to one monitoring system, but it is possible to publish metrics data to multiple monitoring systems by using the **`CompositeMeterRegistry`**:

```
CompositeMeterRegistry composite = new
CompositeMeterRegistry();
```

The metrics generation activity always starts with a registry, regardless of whether it is a single or composite one. Next, let us

check how registries use meters to capture metric data.

Meters and tags

Micrometer supports a set of distinct metrics that are defined as meters. Some of the most used meters are the counter, timer, gauge, distribution summary, and timer. Every meter has a name and a group of dimensions, known also as tags. Tags enable us to put more context into the metric data that is being captured. For example, we can have a counter meter named **`http.request`** with tags like the HTTP method, browser, and operating system. Consider the following example:

```
SimpleMeterRegistry meterRegistry = new
SimpleMeterRegistry();

var httpMethod = "GET"

Counter.builder("http.request")

    .tag("HTTP Method", httpMethod)

    .register(meterRegistry)

    .increment();
```

This code creates a counter meter with a tag that specifies the request's HTTP method. Here, we manually set it to GET, but in a real-world scenario, we get the HTTP method from the underlying technology handling the HTTP request. The more tags we add, the more we increase the metric dimensionality, which makes the metric more valuable because we can slice and dice through the metric tags.

Counters

There are scenarios where we are interested in knowing the frequency at which something occurs inside the system. Imagine a web application that receives HTTP requests at different endpoints. We can use a counter to measure the rate at which the application processes HTTP requests. A counter corresponds to a positive number that can be incremented by one or any other arbitrary number. The following is how we can create and use a counter:

```
SimpleMeterRegistry meterRegistry = new
SimpleMeterRegistry();
```

```
Counter httpRequestCounter = Counter
        .builder("http.request")
        .description("HTTP requests")
        .tags("Source IP Address", "Operating
System")
        .register(meterRegistry);
httpRequestCounter.increment();
httpRequestCounter.increment();
httpRequestCounter.increment();
System.out.println(httpRequestCounter.count()); //
3
```

Micrometer provides a builder that lets us intuitively construct the **`Counter`** object. Note that we are setting the counter meter name as **`http.request`**, along with other data, including the description and the meter tags. Ultimately, we need to pass the **`meterRegistry`** reference object used to create the counter meter. A metric is generated when calling the increment method from the **`Counter`** object. Every time the increment method is called, the count metric number is incremented by one.

Gauges

We use gauges whenever we want to measure the size of a collection of things that can increase or decrease in a system. For example, we can check how many threads are active in the system. The number of threads can increase or decrease depending on the moment the metric is captured. The following code shows how we can create a gauge meter:

```
AtomicInteger totalThreads = meterRegistry.gauge(
        "Total threads",
        new
AtomicInteger(ManagementFactory.getThreadMXBean().getThr
);
totalThreads.set(ManagementFactory.getThreadMXBean().get
```

Instead of using the builder as we did for the counter, we create the gauge meter directly in the **`meterRegistry`** object by providing the value the gauge measures. In the example above, we provide the total number of threads in the system. We wrap it within an **`AtomicInteger`** because we cannot use primitive numbers or object numbers from **`java.lang`** because they are immutable, which would not allow us to update the gauge value after we have defined it for the first time.

Timers

In some situations, we want to know how long an operation takes to complete, and timers are the measure we can use for those situations. Timers are particularly helpful in identifying if some system behavior is taking longer than expected to finish, for example. The following code lets us use the timer meter:

```
SimpleMeterRegistry meterRegistry = new
SimpleMeterRegistry();

Timer durationTimer =
meterRegistry.timer("task.duration");

timer.record(() -> {

    try {

          TimeUnit.SECONDS.sleep(5);

     } catch (InterruptedException _) {

     }

});

System.out.println(durationTimer.totalTime(TimeUnit.SECO
// 5 5.000323186
```

The **`Timer`** interface has a method called **`record`**. As we did in the example above, we can put the system operation we want to measure inside the **`record`** method by placing the call to **`TimeUnit.SECONDS.sleep(5)`**. After the execution, we could check that the timer metric recorded five seconds as the time to execute the task.

Distribution summaries

A recurrent use case for distribution summaries is when we want to measure the file size an application provides for download. Similarly, we can use a distribution summary to track the payload size of upload requests handled by the application. The following is how we can create and use a distribution summary meter:

```
SimpleMeterRegistry registry = new
SimpleMeterRegistry();

var fileSize = 243000.81;

DistributionSummary responseSizeSummary =
DistributionSummary

        .builder("file.size")

        .baseUnit("bytes")

        .register(registry);

responseSizeSummary.record(fileSize);

System.out.println(responseSizeSummary.totalAmount());
// 243000.81
```

Although not mandatory, setting the **`baseUnit`** to express which size unit you intend to track is recommended.

Now that we know how Micrometer works, let us learn how to use it together with Spring Boot to produce application metrics.

Using Micrometer and Spring Boot to implement metrics

Spring Boot provides full support and seamless integration with Micrometer, allowing us to export metrics to various monitoring tools. When used together with Spring Boot, Micrometer metrics are managed by the Spring Boot Actuator component, which exposes data indicating how healthy the application is. The Actuator is often used with probe mechanisms, like the readiness and liveness probes from Kubernetes, to health check the application state through endpoints exposed by the Actuator.

In this section, we explore how to set up Spring Boot with Micrometer to implement a file storage system that lets users upload and download files stored on an in-memory database. To

understand how metrics can be used in real-world scenarios, we instrumentalize parts of the file storage system with Micrometer metrics like counter, timer, and distribution summary.

Let us start by setting up the Spring Boot Maven project with Micrometer dependencies.

Setting up the Maven project

To make Micrometer work with Spring Boot, we must provide specific dependencies from both projects. The following is how we can configure the dependencies in a Maven's **`pom.xml`** file:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
        <artifactId>spring-boot-starter-web</
artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
        <artifactId>spring-boot-starter-data-jpa</
artifactId>
    </dependency>
    <dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-
prometheus</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</
groupId>
        <artifactId>spring-boot-starter-actuator</
```

```
artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

We use the **`spring-boot-starter-web`**, **`spring-boot-starter-data-jpa`**, and **`h2`** dependencies to implement the file storage system that exposes a RESTful API and persist file content in an H2 in-memory database. Micrometer provides dependencies with support for specific monitoring systems. Such dependencies are necessary because there is no standard for metrics format across different monitoring systems. Micrometer lets developers choose which dependencies they want based on the monitoring tools with which they intend to export the metrics. Here, we use the **`micrometer-registry-prometheus`** dependency because our system will export metrics compatible with Prometheus. Finally, we declare the **`spring-boot-starter-actuator`** dependency responsible for exposing the metrics endpoint that monitoring tools can use to get application metrics.

With the dependencies properly configured, we can proceed to configure Spring Boot and Micrometer.

Configuring Spring Boot and Micrometer

The configuration takes place in the **`application.yml`** file:

```
spring:
  servlet:
    multipart:
      max-file-size: 10MB
      max-request-size: 10MB
  datasource:
```

```
    url: jdbc:h2:mem:mydb
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    database-platform:
org.hibernate.dialect.H2Dialect


management:
  metrics:
    enable:
      all: false
      file: true
  endpoints:
    web:
      exposure:
        include: Prometheus
```

We set the properties **`max-file-size`** and **`max-request-size`** to **`10MB`** because the Spring Boot default configuration is **`1MB`**, which is insufficient because we intend to upload files bigger than that. The **`datasource`** property block contains the configuration that defines the H2 in-memory database mode.

The management block is where we define the properties that govern the behavior of the actuator and Micrometer. By default, Spring Boot collects many technical metrics from the system, mainly from the JVM. For simplicity's sake, we are not interested in them, so we disable these metrics by setting **`management.metrics.enable.all`** to **`false`** to turn off all metrics from the system. However, we cannot leave the configuration that way; otherwise, no metric will be captured. To solve it, we set the property **`management.metrics.enable.file`** to **`true`**, making Spring Boot capture all metrics with the word **`file`** at the beginning of their names. Finally, we define how

metrics should be exposed by setting **`endpoints.web.exposure.include`** to **`prometheus`**. That does not mean Spring Boot will connect to a Prometheus instance to export Micrometer metrics. Instead, it means that the Micrometer will generate metrics compatible with Prometheus.

Having correctly defined the dependencies and adequately configured the Spring Boot application to work with Micrometer, we are ready to start implementing the file storage system with metrics instrumentalization.

Enabling metrics on the file storage system

Let us create the bootstrap class that starts the Spring Boot application:

```
@SpringBootApplication
public class FileStorageApplication {

    public static void main(String... args) {

SpringApplication.run(FileStorageApplication.class,
args);
    }
}
```

We place the **`@SpringBootApplication`** annotation on top of the **`FileStorageApplication`** class to make Spring Boot use this class to initiate the application.

As our file storage system operates by storing files in a database, it is imperative that we implement a Jakarta entity. This entity will be our key component for interacting with the database. Let us proceed with this task.

Implementing the File entity

Our intent with the **`File`** entity is to store the file name and its content in the database. The following code is how we can implement such an entity:

```
@Entity
```

```
public class File {

    @Id
    private String id;

    private String name;

    @Lob
    @Column(length = 20971520)
    private byte[] content;

    // Constructors, getters, and setters omitted
}
```

The file ID is managed by the application, so we are not using an ID auto-generation mechanism that delegates the ID generation to the database. The attribute **`name`** stores the file name, and the **`content`** stores the file data in byte array format. Note that we are using the **`@Lob`** annotation and specifying the column length to 20 megabytes.

After implementing an entity, we need a repository interface containing operations that let us persist and retrieve files from the database. Let us implement it.

Implementing the File repository

The **`File`** repository handles **`File`** entity objects by persisting and retrieving them from the database. Following is the **`File`** repository interface implementation:

```
@Repository
public interface FileRepository extends
CrudRepository<File, String> {

    Optional<File> findById(String Id);
}
```

The method declaration called **`findById(String Id)`** lets us retrieve **`File`** entities by their respective IDs. We do not need to declare a method for persisting **`File`** entities because the parent interface, **`CrudRepository`**, already provides such a method.

We still need to implement the service and controller classes, but before doing so, let us create the class containing the **`File`** metric builders.

Implementing the File metrics

To instrumentalize our file storage application, we need to provide meters that we can use to measure different aspects of the system. One way to do that is to create a managed bean class with public methods that return meter objects we can use in other application classes. Let us start by defining the initial class structure:

```
@Component
public class FileMetric {

    private final MeterRegistry meterRegistry;

    @Autowired
    public FileMetric(MeterRegistry meterRegistry)
{
        this.meterRegistry = meterRegistry;
    }
    // Code omitted
}
```

The **`FileMetric`** class is annotated with the **`@Component`**, making it a Spring managed bean that can be used in other Spring beans. We use the class constructor to initialize the **`meterRegistry`** class attribute. Remember that **`MeterRegistry`** is an interface with implementations for different monitoring systems. Since we defined the specific Micrometer Maven dependency with support for Prometheus, when we start this Spring Boot application, the **`meterRegistry`** attribute will be initialized with a **`PrometheusMeterRegistry`** type that is a **`MeterRegistry`**

implementation. Next, we implement the methods that return the meters we intend to use in the application:

```
@Component
public class FileMetric {
    // Code omitted
    public Counter requestCounter(String method,
String path) {
        return Counter
                .builder("file.http.request")
                .description("HTTP request")
                .tags("method", method)
                .tags("path", path)
                .register(meterRegistry);
    }

    public Timer fileUploadTimer(String fileName)
{
        return Timer
                .builder("file.upload.duration")
                .description("File Upload
Duration")
                .tags("fileName", fileName)
                .register(meterRegistry);
    }

    public DistributionSummary
fileDownloadSizeSummary(String fileName) {
        return DistributionSummary
                .builder("file.download.size")
                .baseUnit("bytes")
```

```
                .description("File Download Size")
                .tags("fileName", fileName)
                .register(meterRegistry);
    }
}
```

The **`requestCounter`** method returns a **`Counter`** object that lets us measure how many HTTP requests are arriving at the file storage system endpoints, which will be implemented soon. Note that we pass the **`method`** and **`path`** parameters used in meter tags. Then, we have the **`fileUploadTimer`** method which returns a **`Timer`** object we use to measure how long the application takes to upload a file. Finally, we have the **`fileDownloadSizeSummary`** method which returns a **`DistributionSummary`** object that we use to measure the size of files users download from the file storage system.

Implementing Micrometer metrics with annotations like **`@Counted`** and **`@Timed`** is also possible. However, the author recommends using the builder approach because not all Micrometer metrics annotations may work out of the box in a Spring Boot application. For example, using the **`@Counted`** annotation with classes annotated with **`@Controller`** requires additional Spring configuration.

After implementing the **`FileMetric`** class, let us implement the service and controller classes with metrics instrumentalization. Let us proceed with the service class.

Implementing the File service

We use the service class as an intermediate layer between the controller and the repository. The service class contains the methods responsible for uploading and downloading files. The following code is how we can define the initial class structure:

```
@Service
public class FileService {

    private final FileRepository fileRepository;
```

```
    private final FileMetric fileMetric;
   @Autowired
    public FileService(FileRepository
fileRepository, FileMetric
    fileMetric) {
        this.fileRepository = fileRepository;
        this.fileMetric = fileMetric;
    }
    // Code omitted
}
```

We inject the **`FileRepository`** and **`FileMetric`** dependencies through the class constructor annotated with **`@Autowired`**. Using the following code, we finish the implementation by providing the methods responsible for uploading and downloading the files:

```
@Service<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
</dependency>
public class FileService {
    // Code omitted
    public void uploadFile(File file) {

fileMetric.fileUploadTimer(file.getName()).record(
                () -> fileRepository.save(file)
        );
    }

    public Optional<File> downloadFile(String id)
{
```

```
        return fileRepository.findById(id);
    }
}
```

The essential point to pay attention to here is the usage of the **`fileUploadTimer`** to measure how long the **`uploadFile`** method takes to save the file in the database when it calls the lambda expression **`() -> fileRepository.save(file)`** wrapped inside the **`record`** method.

Let us finish the file storage system implementation by creating the controller class and using the counter and distribution summary meters.

Implementing the Controller class

We implement the controller class to define the RESTful API endpoints responsible for receiving HTTP requests for uploading and downloading files. Following is the initial class structure:

```
@RestController
public class FileController {

    private final FileService fileService;
    private final FileMetric fileMetric;

    @Autowired
    FileController(FileService fileService,
FileMetric fileMetric) {
        this.fileService = fileService;
        this.fileMetric = fileMetric;
    }
    // Code omitted
    private void incrementRequestCounter(String
method, String path) {
        fileMetric.requestCounter(method,
path).increment();
```

```
    }
    private void recordDownloadSizeSummary(File
file) {

    fileMetric
    .fileDownloadSizeSummary(

file.getName()).record(file.getContent().length);
    }
}
```

We inject the **`FileService`** and **`FileMetric`** dependencies through the class constructor annotated with **`@Autowired`**. Next, we define the **`incrementRequestCounter`** method using the **`requestCounter`** method from the **`FileMetric`** to increment the counter metric. We do something similar with the **`recordDownloadSizeSummary`**, which records the file content size.

We are now ready to implement the methods representing the HTTP endpoints for uploading and downloading files. Let us start with the upload endpoint:

```
@RestController
public class FileController {
    // Code omitted
    @PostMapping("/file")
    private String
uploadFile(@RequestParam("file") MultipartFile
file)
    throws IOException {

incrementRequestCounter(HttpMethod.POST.name(), "/
file");

        var fileToUpload = new
File(UUID.randomUUID().toString(),
```

```
        file.getOriginalFilename(),
file.getBytes());
        fileService.uploadFile(fileToUpload);
        return fileToUpload.getId();
    }
    // Code omitted
}
```

Right at the beginning of the method body, we call the method **`incrementRequestCounter(HttpMethod.POST.name(), "/file")`** defined previously. We pass the **`HttpMethod.POST.name()`** and the `"/file"` path to it as the metric tags. The counter meter will be incremented whenever a POST request arrives at this endpoint.

Next, we implement the endpoint that handles file downloads:

```
@RestController
public class FileController {
    // Code omitted
    @GetMapping("/file/{id}")
    private ResponseEntity<Resource>
downloadFile(@PathVariable String id) {

incrementRequestCounter(HttpMethod.GET.name(), "/
file/"+id);

        var file = fileService.downloadFile(id);

        if (file.isEmpty()) {
            return
ResponseEntity.notFound().build();
        }

        recordDownloadSizeSummary(file.get());
```

```
        var resource = new
ByteArrayResource(file.get().getContent());
        return ResponseEntity.ok()

.contentType(MediaType.APPLICATION_OCTET_STREAM)

.contentLength(resource.contentLength())

.header(HttpHeaders.CONTENT_DISPOSITION,

ContentDisposition.attachment()

.filename(file.get().getName())

.build().toString())
                .body(resource);
    }
    // Code omitted
}
```

The **`downloadFile`** method tracks two metrics: first, a request counter metric by calling **`incrementRequestCounter(HttpMethod.GET.name(), "/file/"+id)`** and second, a download size summary metric through **`recordDownloadSizeSummary(file.get())`**. Requests to the **`/file/{id}`** endpoint will trigger a file download in the web browser.

Next, let us check how we can make requests to test our application and visualize the metrics captured by the *Micrometer*.

Compiling and running the sample project

In this section, we compile and run the file storage system we implemented throughout the previous chapter.

You can clone the application source code from the GitHub

repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2010**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

Execute the following command to compile the application:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the following command:

```
$ java -jar target/chapter10-1.0-SNAPSHOT.jar
```

Next, we go over the steps to generate and visualize application metrics:

1. With the application running, you can use the following command to upload a file:

   ```
   $ curl --form file='@random.txt' localhost:8080/file

   69d67cea-4022-4262-a217-a89b9f52e57b
   ```

   The previous command returns a file ID you can use to download a file in your browse with the URL **http://localhost:8080/file/{fileId}**. You need to replace `{fileId}` with the ID provided by the response you got after uploading the file, as shown in the following figure:

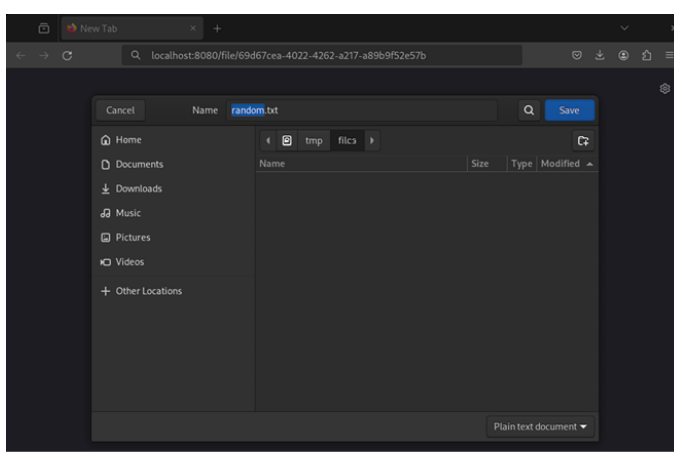

Figure 10.1: *Downloading a file from the file storage system*

Note that the file name **`random.txt`** is the same file name provided when the file was uploaded.

2. We can access the URL **http://localhost:8080/actuator/prometheus** to visualize the application metrics, as shown in the following figure:

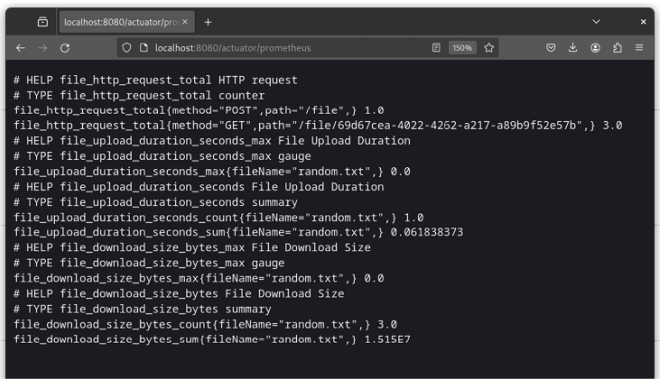

Figure 10.2: *Checking metrics produced by the application*

Looking at the metrics in the image above, we can state that the **`/file`** endpoint received one POST request captured by the following metric:

```
file_http_request_total{method="POST",path="/file",} 1.
```

3. On the other hand, we can check that the **`/file/69d67cea-4022-4262-a217-a89b9f52e57b`** endpoint received three requests measured by using the following metric:

```
file_http_request_total{method="GET",path="/file/69d67c
```

The following two metrics captured the time to upload the file and the file download size, respectively:

```
file_upload_duration_seconds_sum{fileName="random.txt",
```

```
file_download_size_bytes_sum{fileName="random.txt",} 1.
```

Note that both metrics have the **`fileName="random.txt"`** as their only tag.

Conclusion

We learned in this chapter, how important metrics are to help us understand application behaviors, playing a fundamental role in measuring how well the business rule code solves the problems the application is supposed to solve. We also explored the main components behind the Micrometer. We learned that the most fundamental components are the registry, which enables the exposure of metrics to specific monitoring tools like Prometheus, and the meter, which represents different metric types like counter, gauge, timer, and distribution summary. After grasping the fundamental Micrometer concepts, we implemented a file storage system using Spring Boot and Micrometer, where we had the chance to implement metrics to measure the HTTP requests to the application endpoints, the time required to upload a file, and the size of downloaded files.

This chapter focused on how Micrometer can be used to generate application metrics. The approach to how such metrics can be scrapped is explored in the next chapter, where we will learn how to use Prometheus and Grafana to scrap application metrics, enabling us to create helpful monitoring dashboards and set alerts. We will learn how to capture and use application metrics with Prometheus after integrating them with Grafana. We will also explore the features provided by Grafana to create dashboards.

Chapter 11

Creating Useful Dashboards with Prometheus and Grafana

Introduction

Over the last chapters, we have been exploring how essential monitoring tools and techniques are to clarify how well a software system is running. Such tools and techniques are fundamental in providing helpful input through metrics, traces, and logs with details describing system behaviors. Metrics, in particular, represent one of the cornerstones of monitoring practices that shed light on the inner workings of applications' operations. In today's world, where the number of applications and the volume of metrics they produce is bigger than ever, a monitoring solution capable of capturing, storing, and serving large amounts of metrics data is essential to ensure efficient monitoring of highly complex and demanding applications. We have Prometheus as the solution that

can help us tackle such a monitoring challenge.

Complementing Prometheus in a frequently used technology monitoring stack is Grafana, a powerful tool that lets us build beautiful and helpful dashboards using metrics from many supported systems, including Prometheus. In this chapter, we explore how to use Grafana and Prometheus to monitor Java applications.

Structure

The chapter covers the following topics:

- Capturing application metrics with Prometheus
- Integrating Prometheus with Grafana
- Creating Grafana dashboards with application-generated metrics
- Compiling and running the sample project

Objectives

By the end of this chapter, you will know the Prometheus architecture and its fundamental concepts, giving you the essential knowledge to introduce Prometheus as a monitoring tool to capture metrics from your Java applications. Having comprehended the core Prometheus concepts, you will learn how to apply them in a real-world monitoring setup where Prometheus captures metrics from a Spring Boot application and serves them to Grafana, which provides a helpful dashboard on top of those metrics. You will also learn how to use Prometheus metrics to trigger alerts using *Alertmanager*.

Capturing application metrics with Prometheus

Released first in 2012, Prometheus is an open-source monitoring and alerting software that works on top of system-generated metrics. We can use Prometheus to monitor the infrastructure of where applications are running by checking, for example, how much CPU, memory, and disk are being used. We can also use Prometheus to monitor application behaviors by capturing metrics provided by instrumented applications. It has been used by many

organizations worldwide, seeking a robust monitoring solution that can be easily integrated into cloud-native environments.

Prometheus has gained popularity due to its simple yet powerful features. It allows us to store and query metrics data in a time-series database. Prometheus also relies on an alerting mechanism called **Alertmanager**, which lets us trigger alerts in various places, including email, phone, and chat systems.

Prometheus is a prevalent component of most enterprise Java projects, supported by well-known frameworks, including Spring Boot, Quarkus, and MicroProfile. Comprehending how Prometheus works and how it can be integrated to monitor existing and new applications is critical for anyone interested in taking their monitoring machinery to the next level. Anyone running Java applications in production without any monitoring is running into a severe risk of missing crucial system information that could be used to avoid unwanted things or prevent something wrong from becoming worse. That is why instrumenting applications with metrics and employing Prometheus to capture those metrics is the first step to mitigating the risk of missing critical system behavior information.

We start our Prometheus exploration by learning about its architecture and how it works with other systems through exporters that expose metrics that dashboard systems like Grafana can consume. After learning the Prometheus architecture, we will install and configure it, enabling us to explore PromQL, a query language that lets us get answers from the metrics stored by Prometheus.

Learning the Prometheus architecture

The Prometheus architecture's main component is the server process that scrapes metrics from monitoring targets. It is worth noting that Prometheus is a pull-mode monitoring tool, which means it pulls monitoring data from the predefined systems through its configuration. The monitoring data, stored as metrics in a time-series database, is made available through HTTP API endpoints that alert and dashboard systems can use. In the following figure, we have a high-level representation of the Prometheus architecture:

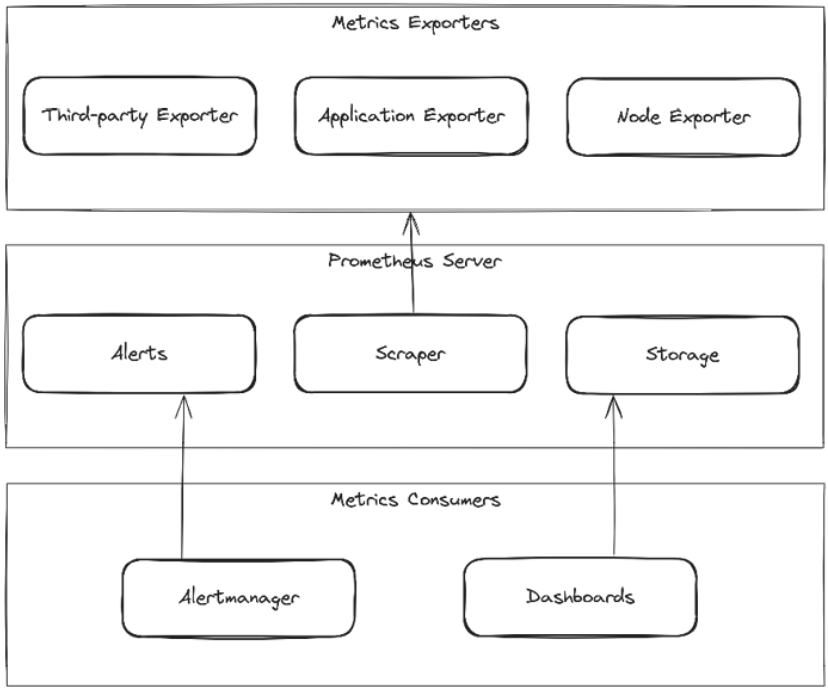

Figure 11.1: *The Prometheus architecture*

Starting at the top in the figure above, the metrics exporters provide metrics data that the Prometheus server scraps. Metric data is stored in the Prometheus storage, allowing us to trigger alerts through the Alertmanager and create dashboards. Let us assess each component of the architecture.

Metrics exporters

Exporters are the architecture components responsible for making metrics available to Prometheus. There are three types of exporters we will cover, as follows:

1. The third-party exporter is used when one wants to get metrics from a system that has no control over its source code. Suppose your application stores data in a MySQL database, and you want to get metrics from it. As you have no control over the MySQL source code, you must rely on an exporter provided by a third party that provides MySQL metrics.

2. The application exporter provides metrics generated by instrumented applications. Contrary to third-party exporters, we have control over the metrics generated by application exporters because we can change the application's code.
3. The node exporter is aimed to expose only machine-based metrics like CPU, memory, and disk usage. It runs as a process in the operating system.

All exporters provide an HTTP endpoint that the Prometheus server uses to pull metrics data. Let us explore what is inside a Prometheus server.

Prometheus server

The Prometheus server is a fundamental architecture component. Its storage is based on a time-series database that persists data directly on the operating system disk. However, remote storage is also supported. The Prometheus server scraps metrics data through a pull mechanism that reaches out to the HTTP endpoints provided by metric exporters. The alerting engine uses metrics data that let us define rules for triggering alerts. Finally, we have metrics consumers covered next.

Metrics consumers

We have two major metrics consumers: the Alertmanager and the dashboard systems. Alertmanager receives alert events from the Prometheus server and triggers notifications through emails, phone calls, and messages in chat systems like *Slack*. Prometheus provides a basic dashboard engine that lets us visually represent the metrics. However, it is recommended and widespread practice to plug in dedicated dashboard systems like Grafana, which we will explore further in this chapter.

Now that we know the Prometheus architecture let us learn how to install the Prometheus server and explore using PromQL, a powerful query language that allows us to perform aggregation on metrics.

Getting Prometheus up and running

Prometheus is compatible with multiple operating systems and CPU architectures. The binary files for the operating system you desire can be found at **https://prometheus.io/download/**. Next, we will

cover how to download, install, configure, and run a Prometheus server in a Linux environment.

Downloading and installing Prometheus

Prometheus installation in a local environment is straightforward because it involves downloading the Prometheus binary, extracting, configuring, and starting it. Next, we cover the steps for Prometheus installation.

1. You download the Prometheus binary through the browser or running the following command on a Linux terminal:

   ```
   $ wget https://github.com/prometheus/prometheus/release
   ```

   The command above downloads the 2.53.0 version, but you can adjust it to get the latest version.

2. Next, we can extract the Prometheus binary file in the same directory we have downloaded it:

   ```
   $ tar xvf prometheus-2.53.0.linux-amd64.tar.gz
   ```

3. After extracting it, you can inspect the extracted files by executing the following command:

   ```
   $ ls prometheus-2.53.0.linux-amd64
   ```

   ```
   console_libraries  consoles  LICENSE  NOTICE  prometheu
   ```

   The following two files deserve our attention: the `prometheus` executable, which starts the Prometheus server, and the `prometheus.yml`, which holds the configuration.

Let us next explore how to provide the initial Prometheus configuration through the `prometheus.yml` file.

Configuring Prometheus

Prometheus configuration is defined through the `prometheus.yml` file that lets us set, for example, the exporter endpoint that the Prometheus server will use to scrape metrics. Next, we will cover the steps to configure Prometheus.

1. Let us consider first how the `prometheus.yml` default configuration looks like:

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to
  evaluation_interval: 15s # Evaluate rules every 15
  # scrape_timeout is set to the global default (10s

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
          # - alertmanager:9093

# Load rules once and periodically evaluate them acc
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endp
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
```

```
      - targets: ["localhost:9090"]
```

The configuration above instructs Prometheus to scrap metrics every 15 seconds, which means Prometheus goes to the configured exporters' endpoints to get metrics data in the mentioned interval. Note also that the alerting configuration is disabled through a hashtag in the line where the alerting target **`alertmanager:9093`** is defined. The most important part of the configuration file is the **`scrap_configs`** block, where Prometheus exporters are defined. The configuration above defines one exporter, the Prometheus server itself, allowing Prometheus to inspect its metrics.

2. There is no need to change the default **`prometheus.yml`** file to get it started, so we can proceed and execute the **`prometheus`** executable file:

```
$ ./prometheus

...

ts=2024-06-23T17:13:41.342Z caller=main.go:1354 level=i

ts=2024-06-23T17:13:41.378Z caller=main.go:1391 level=i

ts=2024-06-23T17:13:41.379Z caller=main.go:1402 level=i

ts=2024-06-23T17:13:41.379Z caller=main.go:1133 level=i

ts=2024-06-23T17:13:41.379Z caller=manager.go:164 level
```

The output above shows that Prometheus is up and running.

3. At this point, you can access the Prometheus user interface through your web browser, as shown in the following figure:

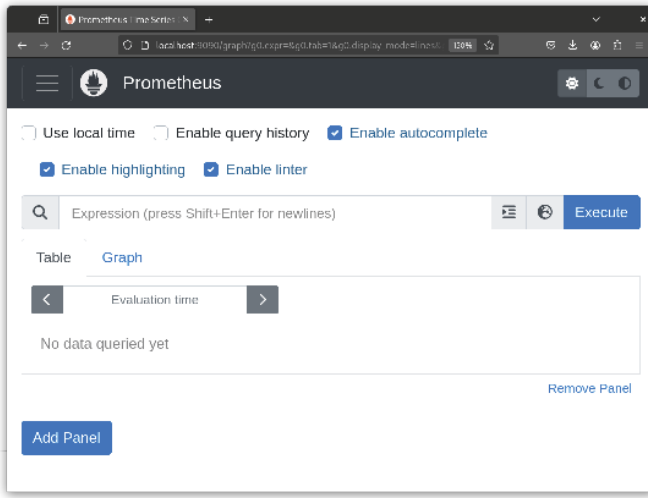

Figure 11.2: *The Prometheus user interface in the web browser*

The figure shows the initial page when you access the URL **http://localhost:9090/** in your web browser.

4. When you click on **Status**, then **Target**, you can see what are the exporter endpoints that Prometheus is connected to, as shown in the following figure:

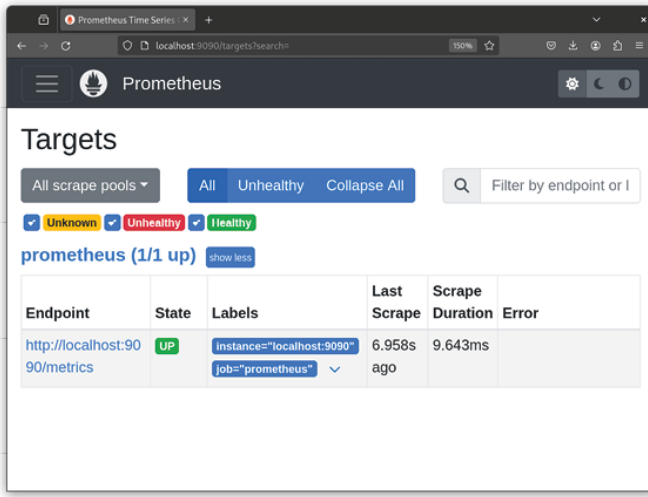

Figure 11.3: *Prometheus user interface in the web browser*

The exporter endpoints in the image above are defined in the

`prometheus.yml` file. The exporter endpoint **http://localhost:9090/metrics** lets us get metrics from Prometheus.

Now that we know how to get Prometheus up and running let us explore **Prometheus Query Language** (**PromQL**), a tool that allows us to query and perform aggregation with metrics data.

Exploring the PromQL

Prometheus has a powerful query language that lets us extract helpful information from raw metric data on how the system behaves. Consider, for example, the usage of the following PromQL expression:

```
file_http_request_total
```

The most basic way of executing a query is by providing the metric name we are interested in, as follows:

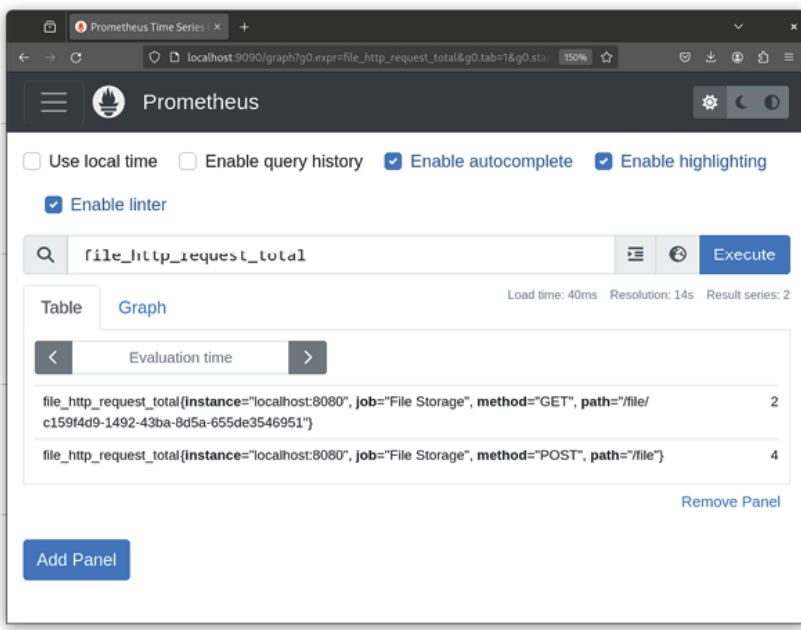

Figure 11.4: *Basic PromQL usage*

By providing the metric name `file_http_request_total` as the query expression, Prometheus returns the metrics data in their raw state. The example above shows that we have the same metric with

different method tags showing how many GET and POST requests arrived at the `/file` endpoint. We can filter out GET metric data with the following expression:

```
file_http_request_total{method="POST"}
```

Which would return something as follows:

```
file_http_request_total{instance="localhost:8080",
job="File Storage", method="POST", path="/file"} 4
```

Number `4` shows how many total POST requests were captured as metric data. With PromQL, we can also check how many requests are arriving per second:

```
rate(file_http_request_total{method="POST", path="/
file"}[5m])
```

The example above uses the rate function, which lets us perform the calculation and tell us how many requests are arriving per second in the last five minutes. Following is the result the above expression can return:

```
{instance="localhost:8080", job="File Storage",
method="POST", path="/file"} 0.03157916897662439
```

The `0.03157916897662439` number tells us the per-second rate at which the system processes POST requests for the `/file` endpoint.

PromQL offers query functions that let us aggregate metric data. In addition to the `rate` function, we can use the `sum` function to accumulate data or the `topk` to get the top aggregation results, for example.

Having covered the fundamentals of how Prometheus works, let us learn how to integrate it with Grafana.

Integrating Prometheus with Grafana

Grafana is an analytics and monitoring solution widely used in integration with other technologies to provide a comprehensive view of monitored systems' behavior. One of its strengths is its ability to visually represent metrics and other monitoring data obtained from systems like Prometheus. In this section, we learn how to configure a simple integration between these two systems using Docker Compose.

1. Through the **`docker-compose.yml`** file, we define a Docker compose configuration for Prometheus and Grafana:

```
version: '3.8'

services:


  prometheus:
    image: prom/prometheus:v2.45.6
    network_mode: host
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/
  grafana:
    image: grafana/grafana:10.4.4
    network_mode: host
    depends_on:
      - Prometheus
```

2. The Prometheus configuration is done through the **`prometheus.yml`** file. Below is how we configure such file:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s


scrape_configs:
  - job_name: "File Storage"
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["localhost:8080"]
```

The `metrics_path` we pass here is the one that is exposed by the file storage application through the Spring Boot actuator. The `localhost:8080` is the URL endpoint where the file storage system will run.

3. We start the Grafana and Prometheus containers up by executing the following command:

```
$ docker-compose up -d

Creating chapter11_prometheus_1 ... done

Creating chapter11_grafana_1    ... done
```

We can confirm that the container setup is working by accessing Prometheus at **http://localhost:9090** and Grafana at **http://localhost:3000**. Next, let us see how to configure Prometheus as a Grafana data source.

Configuring Prometheus as a Grafana data source

When you access the Grafana URL **http://localhost:3000**, you are asked for a username and password. You can use admin for both. Once logged in, you can follow the menu options **Connections**, then **Data sources**. After clicking on the **prometheus** icon, you will find the interface that lets you define Prometheus as the data source, as shown in the following figure:

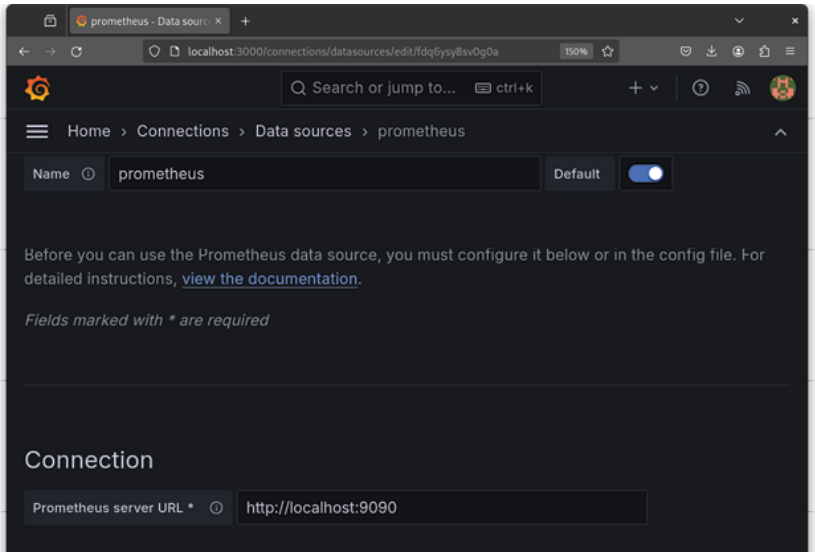
prometheus - Data sourc
localhost:3000/connections/datasources/edit/fdq6ysy8sv0g0a
150%
Search or jump to...
ctrl+k
Home › Connections › Data sources › prometheus
Name
prometheus
Default
Before you can use the Prometheus data source, you must configure it below or in the config file. For detailed instructions, view the documentation.
Fields marked with * are required
Connection
Prometheus server URL *
http://localhost:9090

Figure 11.5: *Defining Prometheus data source on Grafana*

We use the Prometheus URL **http://localhost:9090** as the data source connection configuration. Once we save it, Grafana is correctly integrated with Prometheus.

At this stage, Grafana is actively consuming metrics data produced by the file storage system and exposed through Prometheus. This marks the beginning of our dashboard-building process.

Creating Grafana dashboards with application-generated metrics

In this section, we learn how to use Prometheus and Grafana to monitor a Java application. To help us with the monitoring setup, we rely on the file storage system we developed in the previous chapter. The file storage system produces metrics that let us understand how many requests the system is receiving, the file upload duration, and the file download size. The outcome here is to represent such metrics in a graphical form through a Grafana dashboard. We start by providing Prometheus and Grafana as containers.

Building a Grafana dashboard

The file storage system lets users upload and download files,

capturing metrics based on such operations. We can use a Grafana dashboard to represent how the file storage system handles user requests. To create such a representation, we are building three dashboard visualizations: one to check the number of requests per HTTP method like GET or POST, another to check file upload duration, and the last visualization to verify the download size.

To create a new dashboard, follow these steps: Go to **http://localhost:3000/dashboards**, click on **New**, and then on **New dashboard**. You will be presented with a screen similar to the one as follows:

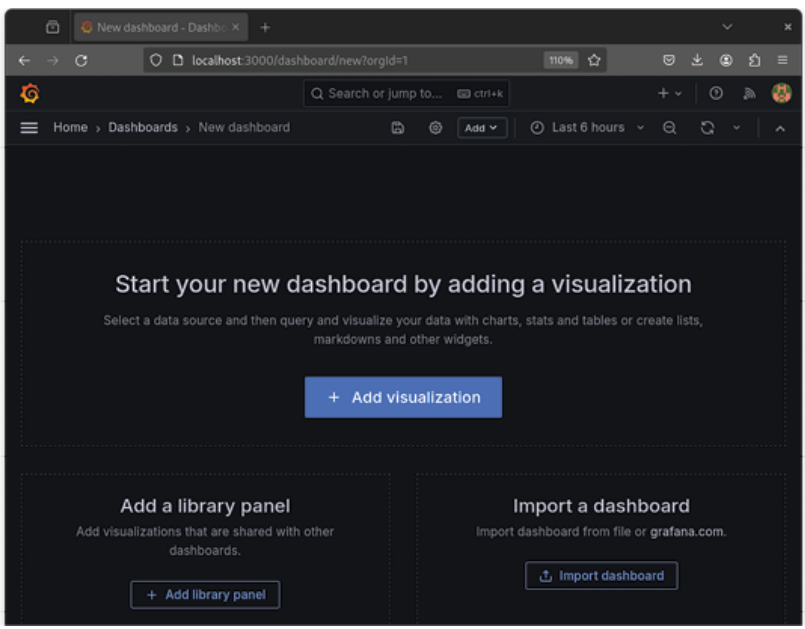
New dashboard - Dashbo
localhost:3000/dashboard/new?orgId=1
110%
Search or jump to...
ctrl+k
Home › Dashboards › New dashboard
Add
Last 6 hours
Start your new dashboard by adding a visualization
Select a data source and then query and visualize your data with charts, stats and tables or create lists, markdowns and other widgets.
+ Add visualization
Add a library panel
Add visualizations that are shared with other dashboards.
+ Add library panel
Import a dashboard
Import dashboard from file or grafana.com.
Import dashboard

Figure 11.6: *New dashboard creation interface*

Grafana dashboards are made of visualizations that display metrics in a graphical way. In the following, we explore how to define the visualizations of the file storage system dashboard.

Visualization for the number of requests per HTTP method

Grafana provides different visualizations we use to display metric data. We use the bar gauge visualization to capture the number of requests per HTTP method, as shown in the following figure:

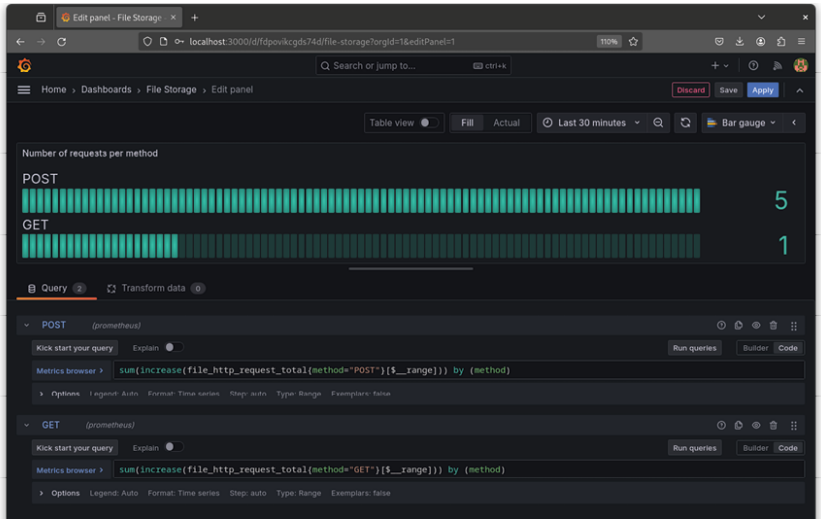

Figure 11.7: *Number of requests per method visualization*

It is possible to define multiple Prometheus queries to get metric data. In the above example, we are defining two queries: one for POST requests and another for GET requests. Note we are employing the **`sum`** and **`increase`** PromQL aggregation functions to calculate how many requests have arrived based on the HTTP method:

```
sum(increase(file_http_request_total{method="POST"}
[$__range])) by (method)
```

The **`increase`** is a Prometheus query function that lets us calculate how a metric increases over a period of time. The **`increase`** function is defined in terms of the time series in the range vector. The time series expresses the time interval we are interested in. It can be expressed as the last five minutes or an explicit time range containing a beginning and ending period. In the example above, we use the **`increase`** to calculate how the **`file_http_request_total`** counter metric increases over a period defined by **`[$__range]`** that represents the time series in the range vector. The **`$__range`** gets replaced by the time interval we set through the Grafana user interface. After calculating the **`increase`**, we sum up all the results based on the method tag (**`POST`** or **`GET`**) defined through the **`by`** keyword.

Next, let us see how to define a visualization to verify the file

upload duration.

Visualization for the file upload duration

We use a time series visualization to display metrics related to the file upload duration, as follows:

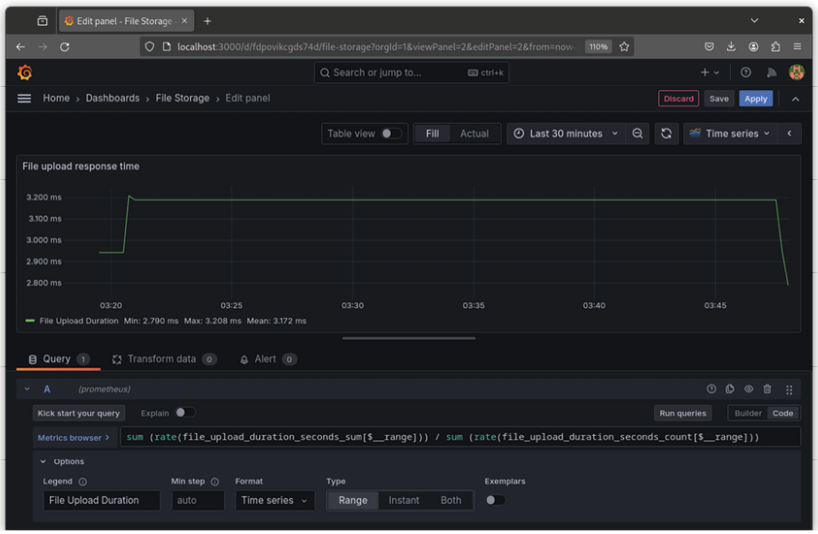

Figure 11.8: *File duration visualization*

The above visualization is based on a single query:

```
sum
(rate(file_upload_duration_seconds_sum[$__range])) /
sum
(rate(file_upload_duration_seconds_count[$__range]))
```

We calculate how long it takes to upload a file by dividing the `sum` aggregation of the `file_upload_duration_seconds_sum` and `file_upload_duration_seconds_count` metrics.

To finish the dashboard configuration, let us see how to create a visualization to check the download size next.

Visualization for the download size

We want to check the top 5 biggest files downloaded in a given time interval. We can accomplish that using a bar gauge visualization, as shown in the following figure:

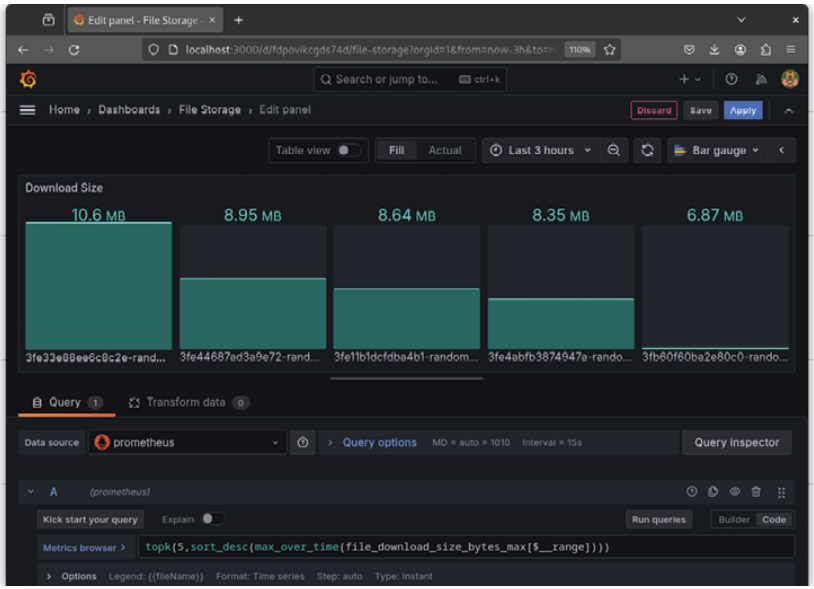

Figure 11.9: *Download size visualization*

The visualization is built using the following query:

```
topk(5,sort_desc(max_over_time(file_download_size_bytes_max
```

We use the **`max_over_time`** function on top of the **`file_download_size_bytes_max`** metric, which is then sorted in a descending way. Finally, we get the top 5 biggest results using the **`topk`** function.

Setting up a Grafana dashboard with meaningful visualizations enhances our visibility over our systems. We can use the dashboard to troubleshoot issues and predict bottlenecks, but we do not want to constantly inspect it to see if some threshold has been reached. For that, we can rely on alerts. Let us explore how we use Prometheus and Alertmanager to trigger alerts next.

Triggering alerts with Alertmanager

Prometheus has a rules engine that lets us use metrics data to trigger alerts based on predefined rules that check if some threshold has been reached. Alertmanager is responsible for receiving the alerts triggered by Prometheus and notifying interested parties through notification channels like email, for example. In this section, we learn how to configure Prometheus and Alertmanager to

trigger alerts based on the metrics generated by the file storage system. Let us start by providing a Docker compose configuration.

Setting up the Alertmanager container

To provide an Alertmanager container, we use the same **`docker-compose.yml`** file we used before to get the Prometheus and Grafana containers:

```
version: '3.8'
services:

  alertmanager:
    image: prom/alertmanager:v0.27.0
    network_mode: host
    volumes:
      - ./monitoring/alertmanager.yml:/etc/
alertmanager/config.yml

  prometheus:
    image: prom/prometheus:v2.45.6
    network_mode: host
    volumes:
      - ./monitoring/prometheus.yml:/etc/
prometheus/prometheus.yml
      - ./monitoring/alert-rules.yml:/etc/
prometheus/alert-rules.yml
    depends_on:
      - alertmanager
# Code omitted
```

We need to adjust the prometheus.yml file to enable the alerting mechanism:

```
# Code omitted
```

```
alerting:
  alertmanagers:
    - static_configs:
        - targets: [ 'localhost:9093' ]


rule_files:
   - "/etc/prometheus/alert-rules.yml"
# Code omitted
```

The alerting configuration block lets us specify the Alertmanager URL **`localhost:9093`** that is used to trigger alerts. With the **`rule_files`** block, we can tell Prometheus where it can find the files containing the alert rules. Let us explore next how we can define Prometheus rules and configure Alertmanager to send email notifications.

Defining Prometheus alerting rule

We can use the file **`alerts-rules.yml`** to set a rule to trigger an alert if the file upload duration time is higher than 2 seconds for 1 minute:

```
# Alert for high File upload duration time
groups:
  - name: File Upload
    rules:
      - alert: HighFileUploadDurationTime
        expr: sum
(rate(file_upload_duration_seconds_sum[1m])) / sum
(rate(file_upload_duration_seconds_count[1m])) > 2
        for: 1m
        labels:
          severity: warning
        annotations:
          summary: "File upload duration time"
```

```
          description: "Time to upload a file: {{
$value }}"
```

The following expression defines the alert rule:

```
sum (rate(file_upload_duration_seconds_sum[1m])) /
sum (rate(file_upload_duration_seconds_count[1m])) >
2
```

An alert is only triggered if the evaluation of the above expression over a minute returns a number higher than 2. When an alert is triggered, Alertmanager is notified and can notify interested parties that Prometheus triggered the alert. Let us see how to accomplish it next.

Defining Alertmanager notification channels

The Alertmanager configuration is done through the **`alertmanager.yml`** file:

```
route:
  receiver: 'mail'
  repeat_interval: 1h
  group_by: [ alerts ]
receivers:
  - name: 'mail'
    email_configs:
      - smarthost: 'smtp.gmail.com:465'
        auth_username: ''
        auth_password: ''
        from: ''
        to: ''
```

The above configuration lets us send email notifications whenever an alert is triggered by one of the Prometheus rules.

Let us finish the chapter by running the file storage system, generating metrics, and testing the alert triggering mechanism with Alertmanager.

Compiling and running the sample project

In this section, we compile and run the file storage system with Prometheus, Grafana, and Alertmanager.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2011**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine. Docker and Docker Compose are also required.

Execute the following command to compile the application:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the following command:

```
$ java -jar target/chapter11-1.0-SNAPSHOT.jar
```

After starting the application, you can bring up the Prometheus, Grafana, and Alertmanager containers with the following command:

```
$ docker-compose up -d
Creating chapter11_alertmanager_1 ... done
Creating chapter11_prometheus_1   ... done
Creating chapter11_grafana_1      ... done
```

Now, we cover the steps to test the Grafana dashboard integrated with Prometheus metrics. We also test the Alertmanager.

1. Access Grafana at **http://localhost:3000** and set the Prometheus data source using the **http://localhost:9090** URL.
2. Then, import the dashboard file located at **`monitoring/file-storage-grafana-dashboard.json`** from the project's repository.
3. Make some requests to generate metrics data:

   ```
   $ curl --form file='@random.txt' localhost:8080/file
   $ curl --form file='@random.txt' localhost:8080/file
   $ curl --form file='@random.txt' localhost:8080/file
   ```

```
$ curl localhost:8080/file/{fileId}

$ curl localhost:8080/file/{fileId}

$ curl localhost:8080/file/{fileId}
```

The `fileId` value comes from executing the first curl commands used to upload the `random.txt` file.

4. Check the File Storage dashboard data on Grafana. The **File Storage** dashboard is as follows:

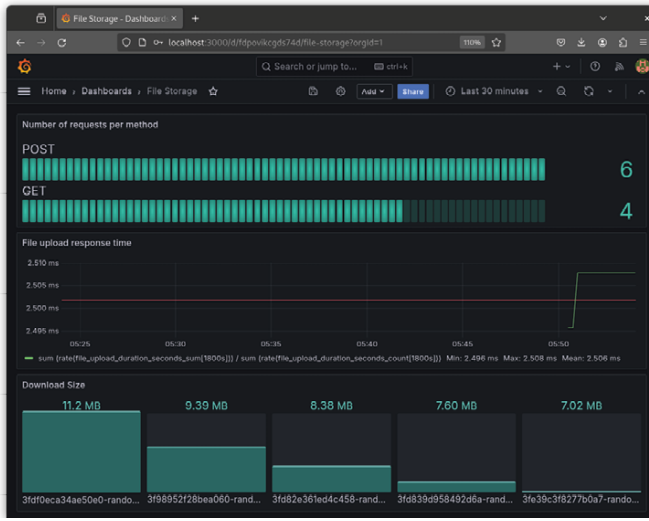

Figure 11.10: *File Storage dashboard on Grafana*

After making many requests to the file storage system, you should see a Grafana dashboard similar to the one shown in the picture above.

5. Execute the following command every 10 seconds for 1 minute:

```
$ curl --form file='@random.txt' localhost:8080/file
```

After executing the command above, you can check the alert captured by *Alertmanager* at **http://localhost:9093**, as shown in the following figure:

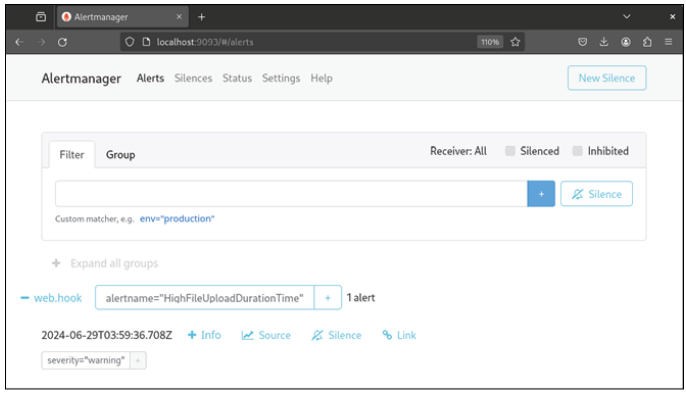

Figure 11.11: *Alert captured by Alertmanager*

We can confirm the alert was successfully captured by checking the alert name **`HighFileUploadDurationTime`**, which is the same as defined in the **`alert-rules.yml`** file.

Conclusion

Storing and serving metrics data is a fundamental responsibility greatly fulfilled by Prometheus, which acts as a monitoring backbone solution by letting us capture application-generated metrics to build meaningful Grafana dashboards and trigger alerts with Alertmanager.

In this chapter, we learned the Prometheus architecture, exploring how metrics are collected through the application, node, or third-party exporters. We also learned how Prometheus stores metrics data and makes them available for alerting and dashboard creation purposes. To explore Prometheus's possibilities further, we integrated it with Grafana by creating visualizations based on the metrics produced by the file storage system. Finally, we configured a Prometheus rule to trigger an alert through the Alertmanager.

In the next chapter, we start a discussion on software architecture by examining a technique called **domain-driven design** (**DDD**), which lets us structure software code in a way that closely represents real-world problems. We will explore DDD concepts such as entities to express identity and uniqueness in a system. We will also learn how to use value objects to enhance the meaning of the domain model.

Chapter 12
Solving problems with Domain-driven Design

Introduction

As a Java developer, most of the software you will see is made to solve business problems. This software represents the processes, rules, and all sorts of things an organization needs to do to stay profitable and fulfill customer expectations. Malfunctioning and bugs in such software may translate directly to financial and reputation damage because most, if not all, business activities depend on the software that enables them.

As the software development industry matured over the years and software systems shifted from mere supporters of business operations to becoming the core actors of business success, developers became more concerned about the practices that allowed them to develop applications that captured business knowledge more accurately. From that concern, one practice called **domain-driven design** (**DDD**) emerged as a software development technique with the main goal of designing applications driven by real-world business problems.

Knowing how to employ domain-driven design is a fundamental

skill for any Java developer interested in building enterprise applications that act as crucial assets for businesses. That is why this chapter covers essential domain-driven design principles and techniques.

Structure

The chapter covers the following topics:

- Introducing domain-driven design
- Conveying meaning with value objects
- Expressing identity with entities
- Defining business rules with specifications
- Testing the domain model
- Compiling and running the sample project

Objectives

By the end of this chapter, you will know how domain-driven design principles such as bounded context and ubiquitous languages help you to model a problem domain that lets you write code that not only works but also serves as an accurate expression of the business operations that are conducted via software to solve real-world problems. By employing techniques such as entities, value objects, and specifications, you will be able to develop better-structured applications by keeping complexity under control and avoiding the so-called big ball of mud systems, where any code change represents a high risk of breaking things.

Introducing domain-driven design

Software projects in an enterprise environment usually start with the organization's desire to solve a perceived problem. The organization assesses the challenges existing and potential new customers face. Then, it attempts to devise solutions to tackle those challenges in the best way possible. In agile-based organizations, we often find a product owner and designer trying to understand with business people what challenges, if solved, will benefit customers and generate profit for the organization.

Once it is clear which kind of problem needs to be solved and the high-level requirements to provide a solution are more or less defined, software developers are summoned to devise a technical path to implement a working software that delivers the value customers are expected to receive. At first glance, it may seem a straightforward path. Still, in reality, the journey is marked by ambiguities, unknowns, and unexpected challenges that developers face when trying to understand what precisely the business needs to fulfill customer expectations.

Understanding business needs can be quite challenging for developers because they are supposed to be experts in the technologies they use rather than in the problem domain from which they want to create a software solution. Knowledge gaps in understanding how a given business operates pose a critical issue because such a weak understanding will translate into an ineffective final software product.

Aware of the fact that weak problem domain knowledge on the part of the software developers represents a considerable risk for the success of enterprise software projects, *Eric Evans* shared through his book *Domain-Driven Design*, published in 2006, ideas explaining how to bridge the gap between software developers and the required problem domain knowledge to design software systems aimed to captured business intricacies in the most cohesive and maintainable way possible. *Evan*'s ideas considerably impacted the software development industry.

Evans conceived a set of principles and techniques that put the problem domain as the main driver for the development of software systems. The problem domain is the specific field in which the business operates, like logistics, banking, and retail, to name some examples. The motivation for having better problem domain knowledge was to make developers think more about business needs than the technologies required to fulfill them. As may be often the case, developers can be tempted to prioritize, for example, which programming language, database, and other technologies they will use rather than trying to understand the problem domain they are dealing with. When taken too far, such temptation leads to a software code driven more by technology than by business. Domain-driven design proposes a change in this mindset with a business-centric perspective on how software can be developed.

Concepts such as the bounded context, ubiquitous language, event storming, and the domain model are corollaries of the domain-driven design. These concepts help us to translate business knowledge into working code. Domain-driven design coding techniques like entities, value objects, and specifications were established to implement the domain model, a tangible materialization based on code, documents, and diagrams that help us understand business problems and how they are solved through software code.

This section explores essential domain-driven design concepts, which will help us implement a domain model in the upcoming sections. Let us start by learning the concept of bounded context.

Bounded contexts

To understand bounded contexts and why mapping them is a fundamental undertaking in domain-driven design, let us consider the business scenario of a personal finance solution. A person usually expects from a personal finance solution the ability to keep track of their expenses by keeping track of how much money they received, where, when, and what they spent their money on. Seeing the money activity through a monthly report is also a valuable capability of such a personal finance solution.

When imagining a software system capable of delivering the personal finance features described previously, we can consider the following three system responsibilities:

- **Money handling**: It is responsible for storing and providing access to money transactions. It also allows for the organization of transactions through user-defined categories like gym, rent, grocery, investments, etc.
- **Report generation**: It contains the rules for generating monthly money activity reports using Excel spreadsheets.
- **File storage**: It provides file storage capabilities, allowing Excel report files to be uploaded and downloaded.

In a traditional monolithic approach, all these three responsibilities would be part of the same application, packed together in the same deployable unit. They would probably be together in the same source code repository. The following figure illustrates the structure

of a monolith application based on the three responsibilities described above:

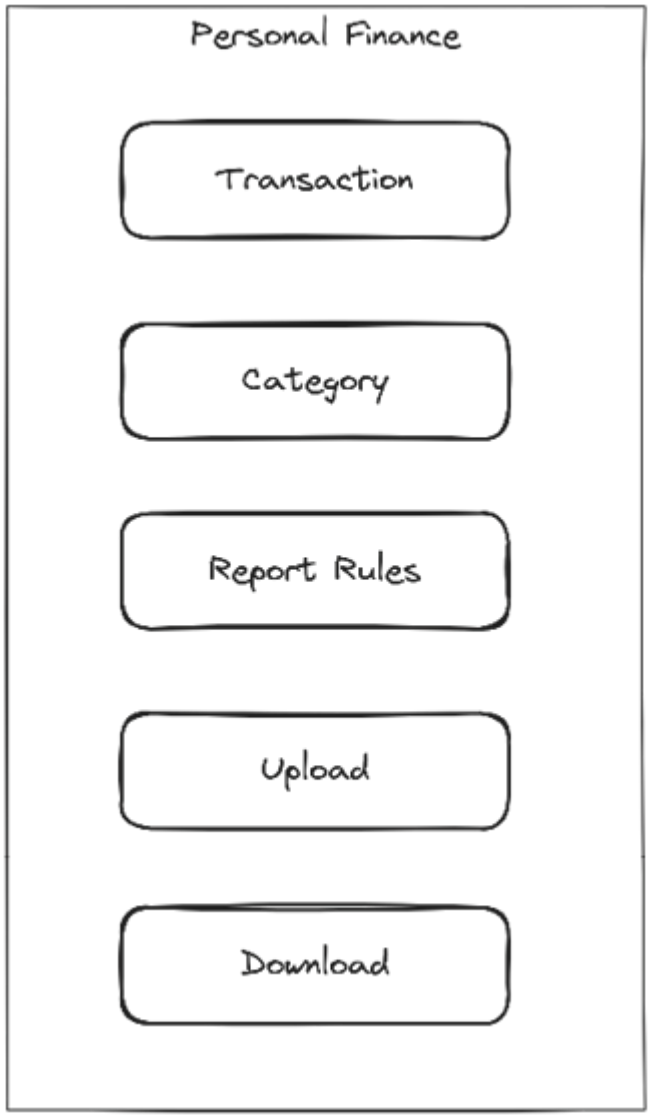

Figure 12.1: *Personal finance monolith structure*

As a monolith system, all system responsibilities related to transactions, categories, report rules, file uploads, and downloads belong to the same context.

We can create a **`Transaction`** class representing money-handling activity. We would use attributes like amount, date, type, and currency to capture the transaction details. Now, imagine we need to represent a transaction in the report engine. We could rely on the same attributes from the **`Transaction`** class of the money handling context but add new attributes like **`totalAmount`**, **`maximumAmount`**, or **`minimalAmount`**. These attributes make sense only in the context of report generation but not in the context of money handling, where they are not used. To avoid this situation, we could create a **`TransactionReport`** class or a new **`Transaction`** class in a different package. However, employing different names would only mask a potential issue in the design: the lack of delimited contexts where a transaction means different things depending on which context it is used.

We can use bounded contexts to solve this problem of a lack of context and define a clear boundary between money-handling and report-generation contexts. In practical terms, that could mean splitting the money handling and report generation responsibilities into separate source code repositories or modules, each containing its version of the **`Transaction`** class based on the context that each system responsibility represents. Going further, the team responsible for the money handling can differ from those responsible for report generation, which would justify the repository split or modularization even more. We can also define a dedicated bounded context for the file download and upload because such capabilities are not directly connected to the problem of handling money or generating reports, as shown in the following figure:

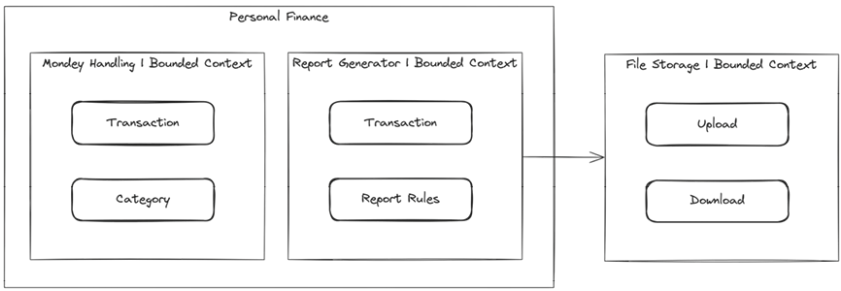

Figure 12.2: *Personal finance with bounded contexts*

Bounded contexts are used to establish a clear delimitation between system responsibilities and remove ambiguities across elements that can have the same name but convey different meanings depending on which context they are situated in.

We used nouns like transaction and category while defining the bounded contexts to describe the system's responsibilities. However, how can we be sure that everyone involved in the personal finance project has the same understanding regarding what those terms mean? We can achieve that by establishing a ubiquitous language. Let us explore it further.

Ubiquitous language

Clear communication is essential for the success of any software project. On the one hand, we have customers expressing their needs. On the other hand, business analysts try to understand those needs and share their learning with product owners and developers. Failure to express or interpret an idea may have serious consequences, as applications may be developed based on faulty thinking. How can we bridge the communication gap across all stakeholders involved in a software project? We can employ the domain-driven design principle called ubiquitous language, which helps us define a set of terms and their meanings that accurately describe a problem domain. These terms must be understood the same way by developers, product owners, designers, business analysts, and any other relevant stakeholders.

The primary benefit of establishing a ubiquitous language is that when a software system is developed based on the terms defined by such language, the application code becomes a source of knowledge of how the business operates. By having the same understanding as domain experts have in the problem domain, developers go one to create application code driven primarily by business needs rather than anything else. There is technology integration with databases and other resources, but it is not the technology choices that drive the code structure; instead, it is the problem domain.

Coming up with an accurate ubiquitous language can be challenging. Sometimes, developers have no clue about the problem domain in which they are supposed to develop a software solution. Such problem domain knowledge usually can be found in the minds

of business analysts or experienced developers who understand how the business operates. Documentation can also be a source of problem-domain knowledge. However, there may be scenarios where people with problem domain knowledge no longer work in the company, and there is no documentation explaining the problem domain. By employing domain-driven design, we can use knowledge-crunching techniques to learn more about the problem domain.

Knowledge crunching can range from reading books on the problem domain area to talking with people who can provide helpful information to understand how a business operates. We use a technique called event storming as a way for knowledge crunching. Event storming is a technique that originated from domain-driven design practices and can yield significant results in understanding the problem domain and building ubiquitous language. Next, we discuss what event storming is and the benefits it can provide to a software project.

Event storming

Most businesses operate based on events representing the interaction of people with business processes. These interactions come from the desire to achieve some outcome carried out by the business process, which is expected to represent a series of steps that, when executed, produce a result. Mapping those business processes and how they work constitutes the major goal of the event storming, which is a workshop session between people who do not know how the business works and people who know. Software developers seeking problem-domain knowledge are the people who need to learn how the business works. On the other hand, those who know about the business process are the so-called domain experts.

Interested parties, like developers and domain experts, sit together to identify domain events and to which business processes those events are associated. Learning about domain events lets one know what must be done to achieve business outcomes. People usually leave these event storm sessions with a better understanding of how problems are solved. Interested parties get the input they need to implement applications that will benefit customers. Domain experts provide their expertise and validate their knowledge by walking

through the steps of the business process.

Aware of the benefits that event storming can provide to help provide clearly defined bounded contexts, the ubiquitous language, and the domain model, we explore how to conduct an event storm session.

Identifying event storm session participants

In an organization that follows agile practices, you may find a product owner, a scrum master, a tech lead, and engineers with back-end, front-end, QA, and DevOps backgrounds. These people are grouped in a product-oriented team, responsible for an entire product or one specific part of a product. Product-oriented teams are usually accountable for the whole life cycle of a software project. By collaborating with business areas, they identify customer requirements, translate them into technical requirements through user stories, and implement, deploy, and actively maintain the software solution. Everyone involved in the software project life-cycle should participate in an event storm session.

Collaboration between business areas and product-oriented teams is essential for the success of a software project. The business areas are usually composed of non-technical people who understand how the business operates. These people, also known as domain experts, must participate in an event-storming session to help those who want to create a software solution but need to learn how the business works.

A facilitator is also necessary to conduct an event storm session. The facilitator is someone acquainted with domain-drive design and event-storming techniques. They guide participants in the right direction by explaining how to identify domain events and map them to their business processes.

Having learned who should be present in an event storm session, let us see how we can prepare it.

Preparing the event storm session

An event storm session produces the best results when conducted as an in-person meeting in a room containing a whiteboard, pens for all participants, and sticky notes in orange, blue, yellow, and green colors. In the following, we see how each stick note color should be

used:

- Orange stick notes describe domain events. We chose orange to emphasize that the domain event is a central element during the event storming exercise.
- Blue stick notes are used for commands. A command specifies an action that triggers a domain event.
- Yellow represents aggregates, which describe the object or data handled by the domain event and command.
- Green colors can represent human actors, such as users, and non-human actors, such as systems.

The standard approach for conducting an event storming session is to rely completely on the whiteboard and physical sticky notes. However, it is also possible to explore a hybrid approach with web collaboration tools. If we go with that approach, then the participants need their laptops during the session.

Asking participants to bring their laptops allows us to explore different forms of collaboration. We can organize an event storm session and put only domain event stick notes on the physical whiteboard. Once we have enough stick notes to describe the business processes we are interested in, we can replicate the stick note representation to a web collaboration tool like *Miro* or *Mural*. We can continue the event storm session in the web collaboration tool by specifying the commands, aggregates, and actors. The benefit of such an approach is that the event storm session results can be referred to and easily adjusted later if necessary.

Let us learn more about domain events, commands, and aggregates.

Domain events

The event storm session starts with the intent to acquire knowledge about how a business process works. Consider, for example, a personal finance solution and how it should work to achieve business outcomes. One of the most critical aspects of such a solution is to enable people to track their expenses by adding their transactions. Based on that, we can define a domain event named **`TransactionAdded`**, as shown in the following figure:

Figure 12.3: *The domain event stick note*

Domain events are always defined as nouns in the past tense describing something as it has already happened. Alright, we have the **`TransactionAdded`** event, but how is it triggered? To do so, we need to define a command. Let us check it next.

Commands

Having identified our first domain event, the **`TransactionAdded`**, we must determine which command triggers such an event. We can solve it by defining the **`AddTransaction`** command, as shown in the following figure:

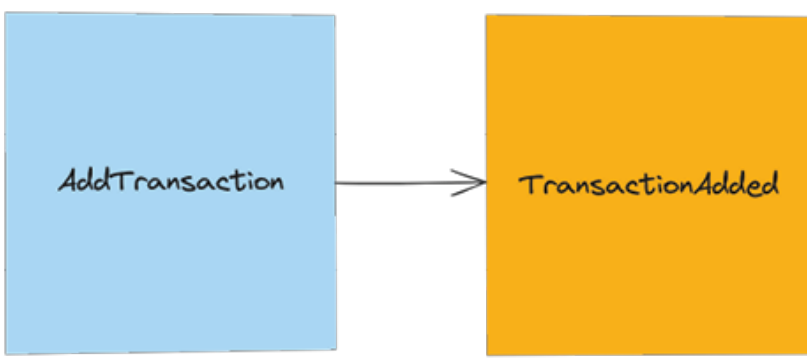

Figure 12.4: *The command stick note*

The **`AddTransaction`** command starts with a verb indicating which action is being carried out to trigger the event. Commands must always be represented with verbs in the present form.

At this stage, we know the domain event and the command that triggers it, but we need to know who is responsible for executing the command.

Actors

Actors play a fundamental role in even storming because it is through them that we can track the source of the existence of domain events. Actors can be defined as humans or non-humans, and their relationship with domain events is mediated through commands as follows:

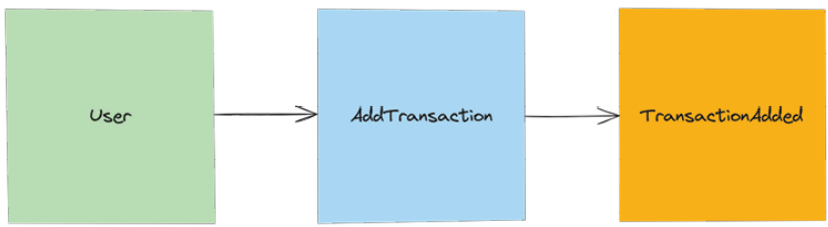

Figure 12.5: *The actor stick note*

Identifying the actors helps us understand who is triggering the events and allows us to explore the motivations behind their interactions with commands and the generated domain events.

Aggregates

We have identified the **`TransactionAdded`** domain event and the **`AddTransaction`** command. The first represents an event that happened, while the second refers to the action generating the event. Domain events and commands represent a connection, an aggregation of activities to fulfill some business outcome. This connection between domain events and commands is described, in our personal finance example, through an aggregate called **`Transaction`**, as shown in the following figure:

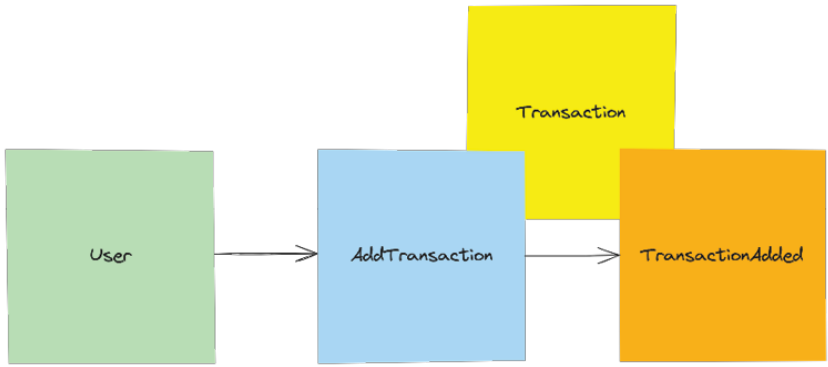

Figure 12.6: *The aggregate stick note*

Think of an aggregate as an entity or data that ties together the command and domain events responsible for enabling the business process. In the example above, the aggregate is positioned between the command and the domain event stick notes.

We can map all the business processes we are interested in by using domain events, commands, actors, and aggregates. Once we have them mapped, the final result will serve as the input for the domain model implementation. In the following section, we will explore what the domain model is.

The domain model

All this discussion around bounded contexts, the ubiquitous language, and event storming we have had so far has significantly increased our capacity to acquire knowledge about the problem domain. We use such knowledge to create the domain model, a tangible representation of the problem domain we want to solve and how that problem is solved. We call it tangible because the domain model usually combines artifacts based on written documents, diagrams, and code. These artifacts share the same ubiquitous language, so the terms used in the application code will have the same meaning as those in documents and diagrams. That is especially helpful because it ensures all software projects' stakeholders have a shared understanding of how the domain model is represented. Non-technical stakeholders can rely on written documents explaining the domain model, knowing that the application code follows the same terminology and meanings.

The domain model is usually expressed through elements such as entities, volume objects, and specifications on the code level. In the next section, we explore using such elements to implement an application using domain-driven design.

Conveying meaning with value objects

In this section, we start developing the personal finances application, a system that lets us keep track of expenses and organize them into categories. A basic personal finances application should allow us to store credit or debit transactions. The transaction itself is the problem domain element we will address soon when discussing entities, but we can start thinking about it now by considering which attributes a transaction may have. Being aware that transactions can be either positive or negative, we can implement the following **`enum`** as a value object:

```
public enum Type {
    CREDIT,
    DEBIT;
}
```

Value objects should be immutable because we use them to give meaning to things inside a domain model. So, to ensure the meaning does not change, we make value objects immutable. Other than having the **`Type`** value object to distinguish different transactions, we can also have a value object to help us provide a meaningful way to express identity:

```
@Getter
@ToString
@EqualsAndHashCode
public class Id {

    private final UUID uuid;

    private Id(UUID uuid) {
        this.uuid = uuid;
```

```
    }

    public static Id withId(String id) {
        return new Id(UUID.fromString(id));
    }
    public static Id withoutId() {
        return new Id(UUID.randomUUID());
    }
}
```

We shorten the code by using the annotations **`@Getter`**, **`@ToString`**, and **`@EqualsAndHashCode`** from the Lombok library. The point of having our **`Id`** class instead of using **`String`** or any other type from the Java standard library is that we can more accurately identify things in a domain model. Note that we have the static method **`withId`** that can be used to create **`Id`** objects based on existing data and the **`withoutId`** that we can use for entirely new data.

Having defined the value objects, we can use them to design the entities of the personal finance application.

Expressing identity with entities

An entity is defined as something with an identity. It is something that we can uniquely identify. In the context of a personal finance system, an account entity can represent the person managing their finances. A person is interested in keeping track of their transactions to know where their money is coming from and where it is going. Every transaction is unique, so it makes sense to have a transaction entity. Just recording the transactions in a single bucket may not be enough; the budgeting practice can produce better results if transactions are categorized. For that, we can have a category entity. Next, we cover the steps to create the **`Transaction`**, **`Category`**, and **`Account`** entity classes.

1. Let us start with the most fundamental entity, the **`Transaction`**:

```
@Builder
public record Transaction (Id id, String name, Doubl

    public static Transaction createTransaction(Acco
    String
    name, Double amount, Type type) {
    // Code omitted
    }

    private static Transaction getTransaction(String
    amount,
    Type type) {
        return Transaction.builder()
                .id(Id.withoutId())
                .name(name)
                .amount(amount)
                .type(type)
                .timestamp(Instant.now())
                .build();
    }

    public boolean addTransactionToCategory(Category
        return category.transactions().add(this);
    }

    public boolean removeTransactionFromCategory(Cat
        return category.transactions().remove(this);
    }
```

```
}
```

We use the **`Id`** value object to define the attribute uniquely identifying a transaction. Every transaction in our personal finance system is unique, making it eligible to be modeled as an entity. The example above relies on Java records to define a **`Transaction`** entity with attributes that let us know its ID, name, amount, type, and timestamp.

In domain-driven design, entities are not seen just as data carriers. Instead, entities represent data and behaviors. In the **`Transaction`** entity above, we define some behaviors, such as adding and removing the transaction from a category, which lets us control which category the transaction will be in. The implementation of the **`createTransaction`** method is not yet available because we want to establish the business rules that need to be followed to allow the creation of a transaction. We will explore it soon when discussing specifications in the next section.

2. A category in our domain model can also be uniquely identified, which makes it eligible to be modeled as an entity. Below is how we can implement the **`Category`** entity:

```
@Builder
public record Category(Id id, String name, List<Tran

    public static Category createCategory(Account ac
    name) {
    // Code omitted
    }

    private static Category getCategory(String name)
        return Category.builder()
                .name(name)
                .id(Id.withoutId())
                .transactions(new ArrayList<>())
```

```
                .build();
    }
   // Code omitted
}
```

We rely on the **`Id`** value object to uniquely identify the **`Category`** entity. We omit the implementation of the **`createCategory`** method because we want to impose business rules to allow the creation of a new category. We will revisit the **`createCategory`** method when discussing the specifications later in this chapter.

3. Transactions and categories can be associated with an account representing a person managing their finances. An account can also be uniquely identified, which makes it eligible to be modeled as an entity. Following is how we can implement the **`Account`** entity:

```
public record Account(Id id, String name, List<Trans

    @Builder
    public Account(Id id, String name, List<Transact
    List<Category>
    categories) {

        this.id = id;
        this.name = name;

        if (transactions == null) {
            throw new RuntimeException("Transaction
            null");
        } else {
            this.transactions = transactions;
```

```
            }
            if (categories == null) {
                throw new RuntimeException("Categories l
                null");
            } else {
                this.categories = categories;
            }
        }
    }
```

The **`Account`** entity is implemented as a record with a constructor with guard checks to ensure lists of transactions and categories are never null. When an account is created for the first time, we expect it to have no transactions or categories. From the code implementation perspective, the constructor accepts empty lists of transactions and categories but cannot accept nulls.

In the domain model, we use entities to capture the data and behaviors that represent the business problem we intend to solve. The entity's behavior can be subjected to constraints that define what can and cannot be done. Such constraints can be expressed through specifications that we will explore next.

Defining business rules with specifications

Business rules are the conditions or prerequisites to fulfill some action. Adherence to these business rules is critical because they ensure the system behaves according to the outcomes the business expects from the software solution. Quite often, we see such rules scattered around the code as if-else statements, defining the conditions the application must meet to proceed with its execution; for those not familiar with the business rules and how the application code handles them, it can be challenging to understand at first glance what those if-else statements mean.

To bring more clarity and enhance the understanding of business

rules through application code, we can use the domain-driven design concept called specification, which is an approach to make business rules more explicit and understandable. Following, we define a specification mechanism using sealed interfaces and abstract classes that we can use later on to implement specifications for the personal finance application.

1. Let us start by defining the **`Specification`** sealed interface:

```
public sealed interface Specification<T> permits Abs

    boolean isSatisfiedBy(T t);

    Specification<T> and(Specification<T> specificat

}
```

   We rely on the Java sealed interface feature to enforce who should implement this interface. The **`Specification`** interface defines the **`isSatisfiedBy`** and **`and`** methods with generic types. We use generics to make the specification flexible and able to deal with any object.

2. Next, we implement the **`AbstractSpecification`** abstract class:

```
public abstract sealed class AbstractSpecification<T

{

    public abstract boolean isSatisfiedBy(T t);

    public abstract void check(T t) throws GenericSp

    public Specification<T> and(final Specification<

        return new AndSpecification<T>(this, specifi

    }

}
```

   Note the usage of **`permits`** with the **`AndSpecification`**, **`DuplicateCategorySpec`**, and **`TransactionAmountSpec`** abstract classes. We will define such classes soon when

providing the specifications for the personal finance application. In **AbstractSpecification**, we define a new abstract method called **check** responsible for performing the business rule validation. The **and** method lets us combine the results with multiple specifications by using the **AndSpecification** abstract class.

3. Following is how we can implement the **AndSpecification** abstract class:

```
public final class AndSpecification<T> extends Abstr

    private final Specification<T> spec1;

    private final Specification<T> spec2;

    public AndSpecification(final Specification<T> s

    Specification<T> spec2) {

        this.spec1 = spec1;

        this.spec2 = spec2;

    }

    public boolean isSatisfiedBy(final T t) {

        return spec1.isSatisfiedBy(t) && spec2.isSat

    }

    @Override

    public void check(T t) throws GenericSpecificati

    }

}
```

The validation occurs inside the **isSatisfiedBy** that evaluates the results of the **spec1** and **spec2**.

Having implemented the specification abstraction, we implement specifications for the personal finance application. For that, we can, for example, define a specification with a business rule that ensures

no transactions with zero amount are entered into the system:

```
public final class TransactionAmountSpec extends
AbstractSpecification<Double> {


    @Override

    public boolean isSatisfiedBy(Double amount) {

        return amount > 0;

    }


    @Override

    public void check(Double amount) throws
GenericSpecificationException {

        if(!isSatisfiedBy(amount))

            throw new
GenericSpecificationException("Transaction value 0
is

            invalid");

    }

}
```

The business rule is implemented inside the **`isSatisfiedBy`**, where we check if the transaction value is greater than zero. Below is how we use the **`TransactionAmountSpec`** in the **`Transaction`** entity:

```
@Builder

public record Transaction (Id id, String name,
Double amount, Type type, Instant timestamp) {


    public static Transaction
createTransaction(Account account, String

    name, Double amount, Type type) {

        var transaction = getTransaction(name,
amount, type);
```

```
        var transactions = account.transactions();

        new
TransactionAmountSpec().check(transaction.amount);

        transactions.add(transaction);

        return transaction;
    }
    // Code omitted
}
```

Whenever a new transaction is created, we add it to a list of transactions. Still, before doing so, we check through the **`TransactionAmountSpec`** to see if the transaction amount is greater than zero. If it is not, then the system throws an exception. We can follow the same approach for the implementation of a specification that ensures the user does not provide duplicate categories:

```
public final class DuplicateCategorySpec extends
AbstractSpecification<Category> {

    private final List<Category> categories;

    public DuplicateCategorySpec(List<Category>
categories) {
        this.categories = categories;
    }

    @Override
    public boolean isSatisfiedBy(Category
category) {
        return categories.contains(category);
```

```
    }

    @Override
    public void check(Category category) throws
    GenericSpecificationException {
        if(isSatisfiedBy(category))
            throw new
GenericSpecificationException("Category already
            exists");
    }
}
```

The **`DuplicateCategorySpec`** has a constructor that receives a list of categories of an account. The specification checks if the new category exists in such a list. If it exists, then it throws an exception. The following is how we can use the **`DuplicateCategorySpec`** in the **`Category`** entity:

```
@Builder
public record Category(Id id, String name,
List<Transaction> transactions) {

    public static Category createCategory(Account
account, String name) {
        var category = getCategory(name);
        var categories = account.categories();

        new
DuplicateCategorySpec(categories).check(category);

        categories.add(category);
        return category;
    }
```

```
    // Code omiited
}
```

The way we use the **`DuplicateCategorySpec`** in the **`Category`** entity is similar to what we did previously in the **`Transaction`** entity. If the validation passes, the category is added to the account's list of categories.

Testing the domain model

The primary benefit of implementing a domain model with domain-driven design techniques is that we can easily test it. Since the problem domain drives the implementation, our code for the domain model should not depend on external resources like databases, making it more straightforward to test.

1. Let us start by testing the **`Account`** entity:

```
public class AccountTest {

    @Test
    public void accountIsSuccessfullyCreated() {
        var name = "testAccount";
        var account = createAccount(name);
        assertEquals(name, account.name());
    }

    private Account createAccount(String name) {
        return Account
                .builder()
                .id(Id.withoutId())
                .name(name)
                .transactions(new ArrayList<>())
                .categories(new ArrayList<>())
```

```
                .build();
    }
}
```

The test above performs a simple test to ensure the **`Account`** entity is created correctly.

2. Next, we test the **`Category`** entity:

```
public class CategoryTest {

    @Test
    public void Given_an_account_exists_create_a_cat
        var name = "testAccount";
        var category = "testCategory";
        var account = createAccount(name);
        assertEquals(0, account.categories().size())

        Category.createCategory(account, category);

        assertEquals(1, account.categories().size())
    }

    @Test
    public void Given_a_category_already_exists_thro
        var name = "testAccount";
        var category = "testCategory";
        var account = createAccount(name);
        Category.createCategory(account, category);
        assertThrows(GenericSpecificationException.c
        Category.createCategory(account, category));
```

```
    }

    // Code omitted

}
```

With the **`Given_an_account_exists_create_a_category`** test, we can confirm whether a new category is created in an existing account. We also test if the specification is really working with the **`Given_a_category_already_exists_throw_exception`** test, which checks if an exception is caught when we try to add an already existing category.

3. To conclude, we test the **`Transaction`** entity:

```
public class TransactionTest {

    // Code omitted

    @Test

    public void Given_an_invalid_transaction_throw_e

        var account = createAccount("testAccount");

        assertThrows(GenericSpecificationException.c

                Transaction.createTransaction(accoun

                tion",

        0.0, Type.DEBIT));

    }

    @Test

    public void Given_a_category_add_and_remove_a_cr

    tion() {

        var account = createAccount("testAccount");

        var category = Category.createCategory(accou

        gory");
```

```
            var transaction = Transaction.createTransact
            "testTransaction", 10.0, Type.CREDIT);

            assertEquals(0, category.transactions().size
            transaction.addTransactionToCategory(categor
            assertEquals(1, category.transactions().size
            transaction.removeTransactionFromCategory(ca
            assertEquals(0, category.transactions().size
        }
        // Code omitted
    }
```

As we did in the **`CategoryTest`**, here in the **`TransactionTest`**, we also test if the specification is working with the **`Given_an_invalid_transaction_throw_exception`** test, which checks if an exception is thrown when the transaction has zero value. With the **`Given_a_category_add_and_remove_a_credit_transaction`** test, we check if the application adds to and removes transactions from a category.

To wrap up, let us compile the personal finance project and run its tests.

Compiling and running the sample project

In this section, we compile and run the tests of the personal finance project we have been exploring over the previous sections.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2012**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

Execute the following command to compile and test the application:

```
$ mvn clean test
```

It should produce an output similar to the following:

```
[INFO]
-------------------------------------------------------

[INFO]  T E S T S

[INFO]
-------------------------------------------------------

[INFO] Running dev.davivieira.entity.CategoryTest

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 0.038 s -- in
dev.davivieira.entity.CategoryTest

[INFO] Running dev.davivieira.entity.AccountTest

[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 0.001 s -- in
dev.davivieira.entity.AccountTest

[INFO] Running dev.davivieira.entity.TransactionTest

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 0.020 s -- in
dev.davivieira.entity.TransactionTest

[INFO]

[INFO] Results:

[INFO]

[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped:
0

[INFO]

[INFO]
-------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO]
-------------------------------------------------------------
```

The output above confirms the tests from **`AccountTest`**, **`CategoryTest`**, and **`TransacationTest`** were successfully executed.

Conclusion

Domain-driven design stands as a reliable approach to designing enterprise applications. By putting business concerns, rather than technology ones, as the main drivers for application development, the domain-drive design approach with concepts like ubiquitous language, bound context, and domain model helps us better understand business problems and how to solve them.

Motivated by the benefits of domain-driven design, we looked at fundamental principles like the ubiquitous language, which fosters shared understanding among developers, business analysts, project owners, and other stakeholders. We also explored the importance of mapping bounded contexts to eliminate ambiguities and clearly define system responsibilities. Furthermore, we discovered event storming, a powerful collaboration technique that brings together developers and domain experts to discuss and gain clarity on the business problems that a software project intends to solve.

Finally, we put these concepts into action by implementing a personal finance application. This practical exercise allowed us to see how entities, value objects, and specifications, all key elements of domain-driven design, can be expressed through Java code.

In the next chapter, we explore how to implement Java applications using layered architecture. We will learn how to develop a Java application using an architecture where the data layer is responsible for data access and manipulation, the service layer provides business rules, and the API layer exposes system behaviors.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 13

Fast Application Development with Layered Architecture

Introduction

Whenever a new software project is started, developers need to decide how the different software components will be structured and interact with each other to fulfill user requirements. Such decisions are made to provide working software running in production in the best way possible. Over the years, developers have been exploring techniques to structure application code that produces working software and let them do so sustainably by identifying and separating concerns in a software system.

One technique, known as layered architecture, has been widely adopted in the enterprise software industry due to its reasonable simplicity and pragmatic approach. When employing layered architecture, it does not take too much to implement it and explain to other team members how it works, which may make it a viable alternative for those wanting to deliver working software faster while keeping, to a certain degree, some order on how the

application code is structured. So, in this chapter, we will explore layered architecture and how we can use it to develop better-structured Java applications.

Structure

The chapter covers the following topics:

- Importance of software architecture
- Understanding layered architecture
- Handling and persisting data in the data layer
- Defining business rules in the service layer
- Exposing application behaviors in the presentation layer
- Compiling and running the sample project

Objectives

By the end of this chapter, you will understand layered architecture by arranging the application code into layers, each holding a specific system responsibility. You will learn how the layered approach helps establish boundaries in the application code, which can contribute to identifying and separating concerns in a software system, positively influencing the overall software architecture. To solidify the concepts explored in this chapter, we will examine the development steps to implement a Java application using layered architecture ideas.

Importance of software architecture

In the software development life cycle, there is a moment when we, as developers, have enough clarity regarding the solution we want to provide to solve some specific problem. In those moments, we have an understanding of user requirements that we judge enough to start developing the code to provide a solution. After having such an understanding, the most important thing we should do in the early stages of any application development is to create a code that solves the problem. Having a dirty, though working code, is much better than having nothing to show. However, once we figure out how to produce this dirty, though working code, we need to start

thinking of ways to polish it, making it better structured and maintainable.

The aim for better structured and maintainable code comes from the fact that we expect the software to change. New features may be introduced, current features may need to change, and bugs will arise. We expect to revisit the code quite often to change it whenever necessary. So, by having the awareness that the software will change in the future, we do not simply want to produce code that only solves the problems we have now, but we also want to have code that lets us easily tackle the issues we may have in the future. That is when we must start thinking about software architecture.

The software architecture we choose is decisive in our capacity to accurately, safely, and quickly introduce code changes whenever needed. A poor software architecture results in what is known as the big ball of mud, an overly complex application code that is hard to grasp but that works. Also, since it works, we, ironically, have a problem. It is a problem because we have software that is useful to its users but a burden to its maintainers. So, whenever a new user requirement arrives, a journey into the unknown begins with the poor developer navigating the intricate and highly complex code base of the big ball of mud software.

How we choose the architecture for the applications we develop is something that, from the author's experience, happens as a conscious effort or spontaneously. When done as a conscious effort, we usually try to assess more or less which kind of application we are developing and how it should evolve. As a result of this assessment, we determine the foundation from which the application will be developed. On the other hand, when we spontaneously choose the software architecture, we make decisions as we go, driven mainly by the desire to have a working code that solves our current problem, sometimes putting the maintainability aspect in the second plane.

The reality is harsh, and sometimes, we do not have the time to spend on careful design. We need to be pragmatic and produce working code now while not neglecting fundamental software architecture aspects that can undermine our ability to change the code in the future. Layered architecture is an approach that helps us

achieve pragmatism, which we will explore further in the next section.

Understanding layered architecture

The layered architecture is a software development technique that helps us structure application code based on the concerns or responsibilities of a software system. To identify those concerns, we can imagine a user making system requests and the steps required to fulfill those requests. The first step can be for the user to interact with a graphical user interface or an API. In this first step, we see a situation where something is presented to those interested in interacting with the application. Presenting something through a graphical user interface or an API can be seen as the presentation concern. In the layered architecture, such a presentation concern can be captured, for example, into the presentation layer.

The second step of a user request may involve data processing, where certain constraints are enforced to determine what the software can and cannot do. These constraints are represented through business rules that establish how the software should behave and how the data provided by the user in step one, the presentation layer, will be processed. Here, we have a part of the system that is not concerned with presenting things, but rather, it is concerned with processing them by applying constraints through business rules that may validate the data provided in the first step. For this part of the system where we process data, we can identify another concern, one that can be part of the business or service layer.

The third and final step of the user request may involve persisting or getting data from a database. The data layer can capture all aspects related to handling database entities.

So far, we have identified the three possible layers: presentation, service, and data layers. It does not mean that all applications can be layered in this way. There is no fixed size on how many layers an application can have. The number of layers depends on our assessment in identifying which application's responsibilities make sense to capture as a layer. That is good because it gives flexibility in defining as many layers as we deem necessary, but at the same time, it can cause trouble if we define more layers than what is

needed. So, prudence is advised when deciding how many layers are necessary for your application. The author usually utilizes a few layers, capturing only the explicit system responsibilities.

Once the layers are defined, we must establish how they will communicate with each other. Let us explore how to do that next.

A layer knows only the next layer

In this approach, the presentation layer would know and interact, for example, directly with the service layer. Every interaction with the data layer would be intermediated through the service layer.

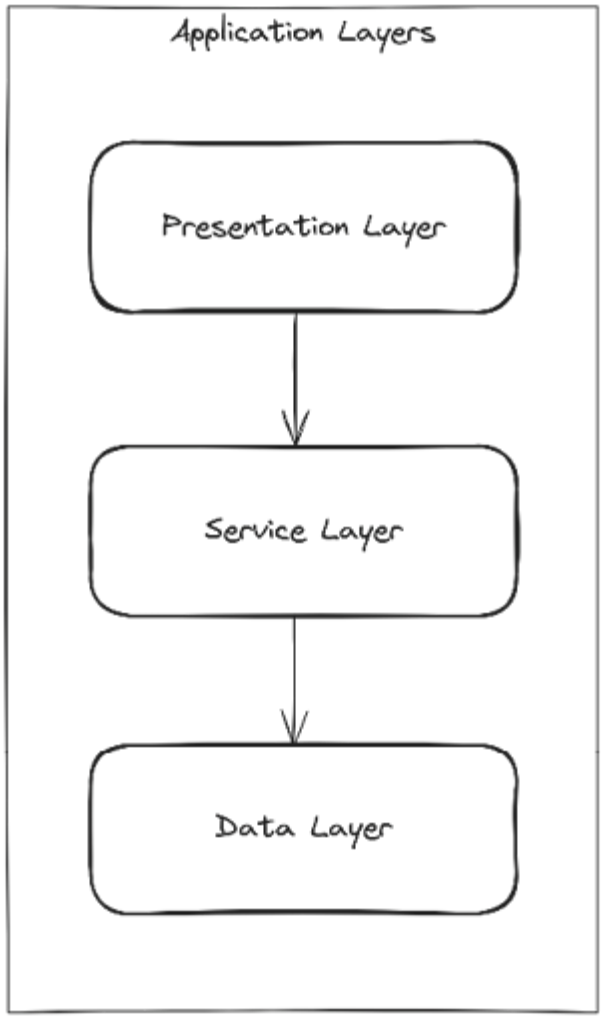

Figure 13.1: *One layer knows only the next layer*

As depicted in *Figure 13.1*, this approach assumes that a higher-level layer knows only the next lower-level layer.

Let us next check an alternative where a layer can interact with other layers.

A layer can know other layers

The presentation layer would be allowed to access, for example, the data layer, as shown in the figure below:

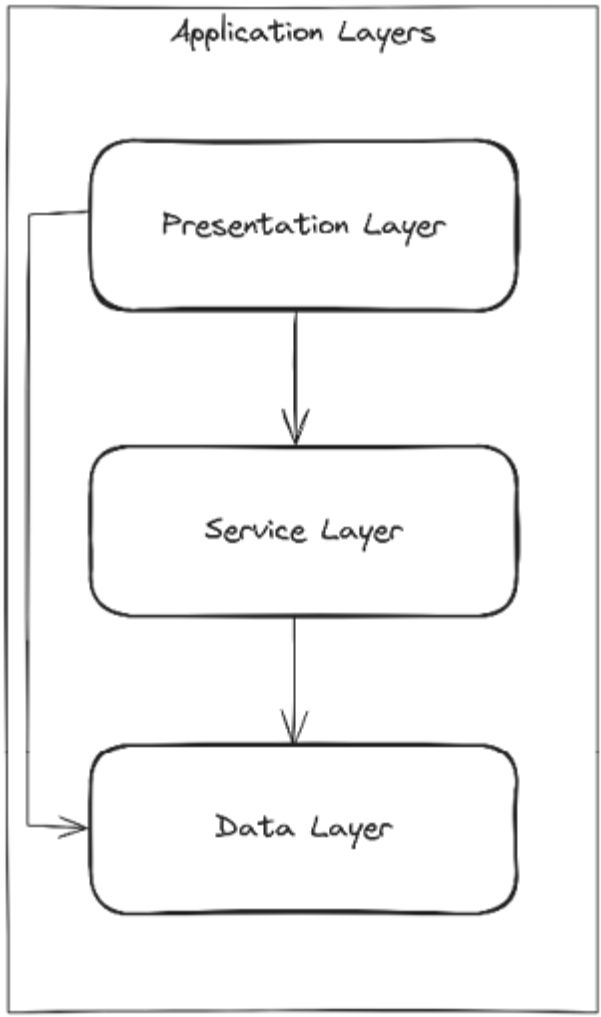

Figure 13.2: *One layer can interact with any low-level layer*

With this approach, we can avoid the situation where a layer is used only as a proxy to access another layer.

Regardless of the layer communication approach, the layer dependency direction must always go downwards because a high-level layer always depends on a lower-level layer. For example, the presentation layer can depend on the service or data layer, but the service layer cannot depend on the presentation layer.

Having grasped the fundamental ideas of layered architecture, let us

now look at their practical application in developing a personal finance system based on the Spring Boot framework. Starting from the layer structure we have discussed so far, which is based on the data, service, and presentation layers, we begin by exploring the role of the data layer in handling data entities and persistence.

Handling and persisting data in the data layer

The personal finance system tracks money spending by storing all user transactions in the database. To enable it, let us implement the **`Transaction`** entity:

```
@Entity
@Builder
@Getter
@AllArgsConstructor
@NoArgsConstructor
public class Transaction {

    @Id
    private String id;
    private String name;
    private Double amount;
    private String type;
    private Instant timestamp;
}
```

We use the **`@NoArgsConstructor`**, **`@AllArgsConstructor`**, **`@Getter`**, and **`@Builder`** annotations from Lombok to make the code more concise. Lombok is a Java library that helps considerably reduce the boilerplate produced by the recurring usage of common language constructs such as constructor declarations and the definition of getters and setters. We use the **`@Entity`** annotation to make the **`Transaction`** class a Jakarta Persistence entity. When declaring the entity class attributes, we must specify which attribute will be used for the entity ID. We do that by placing the **`@Id`**

annotation on the **`id`** attribute.

Next, we can define the repository interface:

```
@Repository
public interface TransactionRepository extends
CrudRepository<Transaction, String> { }
```

Here, we are just extending the **`CrudRepository`** without defining any additional method because we are relying only on the basic database operations that are already provided when the **`CrudRepository`** is extended.

Next, we implement the category entity and its repository.

Implementing the category entity and repository

We intend to let users create categories to group similar transactions. Following, we define the **`Category`** entity class:

```
@Entity
@Getter
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Category {

    @Id
    private String id;
    private String name;

    @OneToMany(cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private List<Transaction> transactions;

    @Override
    public boolean equals(Object o) {
```

```
        if (this == o) return true;
        if (o == null || getClass() !=
o.getClass()) return false;
        Category category = (Category) o;
        return Objects.equals(name,
category.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}
```

Note that we have a **`@OneToMany`** annotation placed above the transactions class attribute. This annotation expresses a one-to-many relationship between a category and one or more transactions. Following this, we override both the **`equals`** and **`hashCode`** methods. The logic we define establishes that **`Category`** objects with the same name are considered equal. Next, we implement the repository interface as follows:

```
@Repository
public interface CategoryRepository extends
CrudRepository<Category, String> { }
```

As we previously did for the **`TransactionRepository`** interface, we extend the **`CrudRepository`** in the **`CategoryRepository`**, hence inheriting all built-in database operations sufficient to handle **`Category`** entities.

Finally, we implement the entity and repository classes to handle accounts.

Implementing the account entity and repository

Every transaction and category belongs to an account. Based on such a relationship, the following code shows how we can implement the **`Account`** entity class:

```
@Entity
@Builder
@Getter
@AllArgsConstructor
@NoArgsConstructor
public class Account {

    @Id
    private String id;
    private String email;
    private String password;

    @OneToMany(cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private List<Transaction> transactions;

    @OneToMany(cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private List<Category> categories;
}
```

There are two one-to-many associations: one that associates an account with transactions and another that associates an account with categories. The idea is that whenever a transaction or category is created, it must be linked to an account.

The following is how we can define the **`AccountRepository`** interface:

```
@Repository
public interface AccountRepository extends
CrudRepository<Account, String> {
    Optional<Account> findByEmail(String email);
}
```

We declare the **`findByEmail`** method to let us retrieve **`Account`** entities based on their email attribute.

With entities, we can map Java classes to database tables, and with repositories, we can establish the operations that let us perform database operations using entities. Entities and repositories comprise the data layer in our layered architecture design.

Let us continue the implementation by defining business rules inside the service layer.

Defining business rules in the service layer

The service layer is where validations and business rules can be enforced on data provided by the user through the presentation layer. Some projects establish the service layer to decouple the business rules from the data layer. We lose flexibility when we have business rules code sitting together with the same code responsible for dealing with the database. For example, suppose we implement business rules based on the underlying database or **object-relational mapping** (**ORM**) technologies, and later, we decide to change such technologies. The business rules may not work with the newer database technology because they were created based on some specific functionality of the older database. Decoupling with layers may help us save refactoring efforts in case of significant application changes like using a new database technology.

The service layer of the personal finance system performs validations to ensure that the data provided by the user does not violate the business rules that govern how transactions and their categories must be handled.

We start by implementing a service class for handling transactions.

Implementing the transaction service

Activities like creating a new transaction, adding a transaction to a category, or removing a transaction from a category can be the responsibility of a transaction service class. We cover the next steps to implement the transaction service.

1. The following code defines the initial class structure:

```
@Service
```

```
public class TransactionService {

    private final CategoryRepository categoryReposit
    private final AccountRepository accountRepositor

    @Autowired
    public TransactionService(CategoryRepository cat
                              AccountRepository acco
        this.categoryRepository = categoryRepository
        this.accountRepository = accountRepository;
    }
    // Code omitted
}
```

We use the **`@Service`** annotation from Spring to make it a managed bean, so we do not need to worry about creating class instances or providing dependencies. We also define **`CategoryRepository`** and **`AccountRepository`** class attributes injected through the **`TransactionService's`** constructor.

2. Continuing with the implementation, we implement methods responsible for creating new transactions:

```
@Service
public class TransactionService {
    // Code omitted
    public void createTransaction(Account account, T
    transactionPayload) throws Exception {
        validateAmount(transactionPayload);
        var transaction = Transaction.builder()
                .id(transactionPayload.getId())
```

```
                .name(transactionPayload.getName())
                .amount(transactionPayload.getAmount
                .type(transactionPayload.getType())
                .timestamp(transactionPayload.getTim
                .build();
        account.getTransactions().add(transaction);
        accountRepository.save(account);
    }

    private void validateAmount(TransactionPayload t
    load)
    throws Exception {
        var amount = transactionPayload.getAmount();
        if(!(amount > 0)) {
            throw new Exception("Transaction value 0
        }
    }
    // Code omitted
}
```

The **`createTransaction`** objects receive as parameters an **`Account`** and a **`TransactionPayload`** object. It uses the payload objects to perform validation using the **`validateAmount`** method that throws an exception if the transaction value is zero or less. If the validation is successful, a new **`Transaction`** object is created and added to the list of existing transactions of an **`Account`** object.

3. We use the **`TransactionPayload`** class to capture data provided by the user:

```
@Getter
```

```
public class TransactionPayload {

    private String id = UUID.randomUUID().toString()
    private String accountId;
    private String name;
    private Double amount;
    private String type;
    private Instant timestamp = Instant.now();
}
```

The payload class plays a vital role by serving as the data carrier class in the service layer. As we will see in the next section, it is also used in the presentation layer to capture user data coming through API requests.

4. To finish the **`TransactionService`** implementation, we define the methods that allow adding and removing transactions from a category:

```
@Service
public class TransactionService {
   // Code omitted
    public boolean addTransactionToCategory(
    Category category,
    Transaction transaction) {
        category.getTransactions().add(transaction);
        categoryRepository.save(category);
        return true;
    }

    public boolean removeTransactionFromCategory(
    Category category,
```

```
        Transaction transaction) {
            category.getTransactions().remove(transactio
            categoryRepository.save(category);
            return true;
        }
    }
```

Both methods, **`addTransactionToCategory`** and **`removeTransactionFromCategory`**, receive a **`Category`** and **`Transaction`** objects used to add and remove transactions from categories.

Let us see now how to implement a service class to handle categories.

Implementing the category service

Other than simply allowing the creation of new categories, we can use the service class to enforce some rules that must be respected before creating the category.

1. Let us start by defining the initial class structure:

```
@Service
public class CategoryService {

    private final AccountRepository accountRepositor

    @Autowired
    public CategoryService(AccountRepository account
        this.accountRepository = accountRepository;
    }
    // Code omited
}
```

We do not persist categories directly into the database.

Instead, we save them through the **`Account`** entity to which they belong. That is why **`AccountRepository`** should be injected as a dependency.

2. Next is the code that lets us create a new category:

```
@Service
public class CategoryService {

    public void createCategory(Account account, Cate
     categoryPayload) throws Exception {
        var category = getCategory(categoryPayload);
        validateCategory(account, category);
        account.getCategories().add(category);
        accountRepository.save(account);
    }

    private Category getCategory(CategoryPayload cat
        return Category.builder()
                .name(categoryPayload.getName())
                .id(categoryPayload.getId())
                .transactions(List.of())
                .build();
    }

    private void validateCategory(Account account, C
    throws Exception {
        var categories = account.getCategories();
        if(categories.contains(category)) {
            throw new Exception("Category already ex
```

```
                count");
            }
        }
    }
```

The `createCategory` receives an `Account` and a `CategoryPayload` object. Before saving the new category into the provided `Account` object, our service class checks, through the `validateCategory` method, if the given category does not already exist. The validation is made by looking at the category's name. No categories with the same name can exist in the same account.

3. Following is what the `CategoryPayload` looks like:

```
@Getter
public class CategoryPayload {

    private String id = UUID.randomUUID().toString()
    private String accountId;
    private String name;
}
```

We use the UUID string defined in the `CategoryPayload` class to persist the `Category` entity into the database.

Next, we finish the service layer development by implementing a service class for handling accounts.

Implementing the account service

Users need to create an account to start using the personal finance system. Such a responsibility can be part of the account service. Let us implement it through the following class:

```
@Service
public class AccountService {
    private final AccountRepository
accountRepository;
```

```
    @Autowired
    public AccountService(AccountRepository
accountRepository) {
        this.accountRepository =
accountRepository;
    }
    public Account createAccount(AccountPayload
accountPayload) throws
    Exception {
        validateEmail(accountPayload);
        var account = Account.builder()
                .id(accountPayload.getId())
                .email(accountPayload.getEmail())

.password(accountPayload.getPassword())
                .categories(List.of())
                .transactions(List.of())
                .build();
        accountRepository.save(account);
        return account;
    }
    private void validateEmail(AccountPayload
accountPayload) throws
    Exception {
        if (!Pattern.matches("^(.+)@(\\S+)$",
accountPayload.getEmail())) {
            throw new Exception("Email format name
is invalid.");
        }
```

```
        if
(accountRepository.findByEmail(accountPayload.getEmail()
            sent()) {
            throw new Exception("Email provided
already exists.");
        }
    }
}
```

The **`AccountService`** is a straightforward service class implementation that relies only on the **`AccountRepository`** class. The **`createAccount`** method receives an **`AccountPayload`** object as a parameter used by the **`validateEmail`** method to ensure the email provided is valid. The new account will be saved in the database if the validation passes. The following is how the **`AccountPayload`** class can be implemented:

```
@Getter
public class AccountPayload {

    private String id =
UUID.randomUUID().toString();
    private String email;
    private String password;
}
```

The **`AccountPayload`** is a data carrier class that captures the user's request data to create a new account.

Having defined the **`TransactionService`**, **`CategoryService`**, and **`AccountService`**, we can now check how the behaviors provided by those classes can be exposed in the presentation layer.

Exposing application behaviors in the presentation layer

In the layered architecture, the presentation layer is commonly seen as a layer associated with graphical user interfaces because of the

idea of presenting something. However, the meaning you give to the layers in your application can be different from the commonly known meaning. In the context of a purely back-end system, the presentation layer can mean the presentation of system behaviors through an API where users and other applications can interact with the system.

Based on this idea that system behaviors can be presented or exposed through an API, we will define the presentation layer for the personal finance system.

Let us start by implementing an API endpoint to handle transactions.

Implementing the transaction endpoint

All possible transaction operations are exposed through a RESTful API endpoint. With such an endpoint, users can request to create transactions and put them into categories.

1. Let us start the transaction endpoint implementation by defining the basic class structure:

```
@RestController
public class TransactionEndpoint {

    private final TransactionService transactionServ
    private final TransactionRepository transactionR
    private final AccountRepository accountRepositor
    private final CategoryRepository categoryReposit

    @Autowired
    private TransactionEndpoint(TransactionService t
    vice,
                                TransactionRepository
                                Repository,
                                AccountRepository acc
```

```
                                        CategoryRepository ca
                                        ) {
            this.transactionService = transactionService
            this.transactionRepository = transactionRepo
            this.accountRepository = accountRepository;
            this.categoryRepository = categoryRepository
        }
        // Code omitted
    }
```

We put the **`@RestController`** annotation from Spring Boot on top of the **`TransactionEndpoint`** class to expose API endpoints. We inject the **`TransactionService`**, **`TransactionRepository`**, **`AccountRepository`**, and **`CategoryRepository`** as dependencies to allow proper transaction management.

2. Having the basic class structure, let us define the endpoints to create and retrieve transactions:

```
@RestController
public class TransactionEndpoint {
    // Code omitted
    @PostMapping("/transaction")
    public void createTransaction(
    @RequestBody TransactionPayload transactionPaylo
    tion {
       var account =
       accountRepository
       .findById(transactionPayload.getAccountId()).
```

```
        transactionService
        .createTransaction(account, transactionPayloa
    }

    @GetMapping("/transactions")
    public List<Transaction> allTransactions() {
        return (List<Transaction>) transactionReposit
    }
    // Code omitted
}
```

The **`createTransaction`** method handles HTTP POST requests at **`/transaction`** containing a JSON payload mapped to the **`TransactionPayload`** class. The payload, based on the data provided by the user, is used to save a new transaction in the system. We use the account ID obtained from the payload to fetch an **`Account`** object. Then, we pass the **`Account`** and **`TransactionPayload`** objects to create the transaction using **`transactionService.createTransaction(account, transactionPayload)`**.

The **`allTransactions`** method is straightforward. It handles HTTP GET requests that retrieve all transactions in the system.

3. Other than allowing the creation and retrieving transactions, the **`TransactionEndpoint`** also contains endpoints that let us add to or remove a transaction from a category:

```
@RestController
public class TransactionEndpoint {
    // Code omitted
    @PutMapping("/{categoryId}/{transactionId}")
    public void addTransactionToCategory(
```

```
      @PathVariable String categoryId,
      @PathVariable String transactionId
    ) {
        var category = categoryRepository.findById(c
        var transaction =
        transactionRepository.findById(transactionId
        transactionService.removeTransactionFromCate
        transaction);
        transactionService.addTransactionToCategory(
        transaction);
    }
    @DeleteMapping("/{categoryId}/{transactionId}")
    public void removeTransactionFromCategory(
      @PathVariable String categoryId,
      @PathVariable String transactionId
    ) {
        var category = categoryRepository.findById(c
        var transaction =
            transactionRepository.findById(transacti
        transactionService.removeTransactionFromCate
        transaction);
    }
     // Code omitted
}
```

To add a transaction to a category, the system expects an HTTP PUT request at the **`/{categoryId}/`**

{transactionId} endpoint through the **addTransactionToCategory** method that receives the **categoryId** and **transcationId** parameters that are used to fetch a **Category** and **Transaction** objects that are used to categorize the transaction through the call of **transactionService.addTransactionToCategory(category, transaction)**. A similar operation occurs for the deletion endpoint that receives an HTTP DELETE request that relies on the call to **transactionService.removeTransactionFromCategory(catego transaction)** to remove a transaction from a category. Note that to change a transaction's category, we must first delete it from the existing category by sending an HTTP DELETE request.

Next, we learn how to implement the category endpoint.

Implementing the category endpoint

As done for the **TransactionEndpoint**, let us start by defining the **CategoryEndpoint** basic structure:

```
@RestController
public class CategoryEndpoint {

    private final CategoryService categoryService;
    private final CategoryRepository
categoryRepository;
    private final AccountRepository
accountRepository;

    @Autowired
    private CategoryEndpoint(CategoryService
categoryService,
                             CategoryRepository
categoryRepository,
                             AccountRepository
accountRepository) {
```

```
        this.categoryService = categoryService;
        this.categoryRepository =
categoryRepository;
        this.accountRepository =
accountRepository;
    }
    // Code omitted
}
```

We inject the **`CategoryService`**, **`CategoryRepository`**, and **`AccountRepository`** classes as dependencies. The **`AccountRepository`** is required because every category is associated with an account, so we use the **`AccountRepository`** to retrieve **`Account`** objects. The **`CategoryEndpoint`** lets users create and list categories, as follows:

```
@RestController
public class CategoryEndpoint {

    @PostMapping("/category")
    public void createCategory(@RequestBody
CategoryPayload
    categoryPayload) throws Exception {
        var account =

accountRepository.findById(categoryPayload.getAccountId(
        categoryService.createCategory(account,
categoryPayload);
    }

    @GetMapping("/categories")
    public List<Category> allCategories() {
        return (List<Category>)
categoryRepository.findAll();
```

```
    }
}
```

The **`createCategory`** method handles HTTP POST requests at the **`/category`** endpoint, creating a new category based on the user-provided JSON payload mapped to **`CategoryPayload`**. On the other hand, the **`allCategories`** method handles HTTP GET requests at the **`/categories`** endpoint, retrieving all available categories.

Having implemented the endpoint classes to handle transactions and categories, we still need to create an endpoint to handle accounts.

Implementing the account endpoint

All transactions and categories of the personal finance system are associated with an account. So, as the first step to using the system, the users must create an account. The following is how we can define the **`AccountEndpoint`**:

```
@RestController
public class AccountEndpoint {

    private final AccountService accountService;
    private final AccountRepository
accountRepository;

    @Autowired
    private AccountEndpoint(AccountService
accountService,
    AccountRepository
    accountRepository) {
        this.accountService = accountService;
        this.accountRepository =
accountRepository;
    }
```

```
    @PostMapping("/account")
    public Account createAccount(@RequestBody
AccountPayload
    accountPayload) throws
    Exception {
        return
accountService.createAccount(accountPayload);
    }

    @GetMapping("/account/{email}")
    public Account getAccount(@PathVariable String
email) throws Exception
    {
        return
accountRepository.findByEmail(email).orElseThrow(()
-> new
        Exception("Account not found"));
    }
}
```

The **`createAccount`** method handles HTTP POST requests at the **`/account`** endpoint that lets users create accounts, while the **`getAccount`** method handles HTTP GET requests, allowing users to get account details by passing the account email address.

At this stage, the personal finance application is implemented using the data, service, and presentation layers. Let us explore next how to compile and run the personal finance application.

Compiling and running the sample project

In this section, we compile and run the personal finance project we have been exploring in the previous sections. As we have exposed an API for the personal finance application, we also explore how to consume such an API in the next section.

You can clone the application source code from the GitHub

repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2013**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

Execute the following command to compile the application:

```
$ mvn clean package
```

Maven will create a JAR file that we can use to run the application by running the command below:

```
$ java -jar target/chapter13-1.0-SNAPSHOT.jar
```

Next, we cover the steps to test the application.

1. You can use the following command to create a new account:

   ```
   $ curl -XPOST localhost:8080/account -H 'Content-type:a
   ```

2. After having an account, we can create some categories for it:

   ```
   $ curl -XPOST localhost:8080/category -H 'Content-type:
   ```

   ```
   $ curl -XPOST localhost:8080/category -H 'Content-type:
   ```

3. The following is how we can create a new transaction:

   ```
   $ curl -XPOST localhost:8080/transaction -H 'Content-ty
   ```

4. To find out the ID of the transaction and category, we can fetch account details with the following request:

   ```
   $ curl localhost:8080/account/john.doe@davivieira.dev
   ```

5. Following is how we can add a transaction to a category:

   ```
   $ curl -XPUT localhost:8080/{CATEGORY_ID}/{TRANSACTION_
   ```

   You can fetch the account details again to confirm that the transaction has been inserted into the desired category.

Conclusion

The ability to group system responsibilities into layers allows one to define boundaries within a system. These boundaries are formed based on our understanding of the steps a software system needs to conduct to fulfill user needs. The layered architecture helps to capture such an understanding into layers that cooperate in

realizing system behaviors. Because of its pragmatic approach, the learning curve to grasp layered architecture is not so high, which makes such architecture a good candidate for fast application development.

Aware of the benefits that layered architecture can provide, in this chapter, we explored the ideas behind designing software systems into layers by implementing the personal finance system employing first the data layer, responsible for abstracting and handling all database interactions, then the service layer in charge of enforcing constraints through business rules that dictate how the software should behave, and finally the presentation layer responsible for presenting through an API or a graphical user interface, the behaviors supported by the personal finance application.

In the next and final chapter, we explore the software design approach called hexagonal architecture, which allows us to develop more change-tolerable applications. We will learn how to arrange the domain model in the domain hexagon, provide input and output ports in the application hexagon, and expose input and output adapters in the framework hexagon.

Chapter 14
Building Applications with Hexagonal Architecture

Introduction

Software development in an environment of constant changes can be challenging. Customers want to receive good service, and organizations strive to provide it with efficient software systems. However, customer needs change. Not only that, the way to fulfill customer needs can also change. After all, enterprises are in an unending quest to produce value in the most inexpensive way possible. Such a landscape presents a formidable challenge for developers who need to solve customer problems now and, at the same time, ensure the systems they are creating can evolve sustainably. By being sustainable, we are referring to the ability to handle software changes gracefully, especially those that deal with fundamental technological dependencies like, for example, database or messaging systems.

We tackle uncertainty by developing software in a change-tolerable way. We can accomplish that by using hexagonal architecture, a

technique that helps develop software so that the technological aspects are entirely decoupled from the business ones. This chapter explores how hexagonal architecture helps create software systems capable of welcoming fundamental technological changes without significant refactoring efforts.

Structure

The chapter covers the following topics:

- Introducing hexagonal architecture
- Arranging the domain model
- Providing input and output ports
- Exposing input and output adapters
- Compiling and running the sample project

Objectives

By the end of this chapter, you will know how to develop Java applications using the hexagonal architecture. You will learn how to define the domain model provided by the domain hexagon. You will also learn how to use input and output ports from the application hexagon to orchestrate system dependencies required to enable behaviors established by the domain hexagon. Finally, you will learn how to use adapters from the framework hexagon to make your system compatible with different technologies.

Introducing hexagonal architecture

Back-end applications are typically designed to work in cooperation with other systems. The interaction with other systems and their underlying technologies is a fundamental aspect of software development because it defines how application dependencies will be provided. We can describe some of the application's dependencies as follows:

- The database technology we use to store the data.
- The message broker system we use to publish and consume messages.

- The scheduler system we use to schedule tasks.

These dependencies can influence how the software is designed. Applications that heavily rely, for example, on a specific relational database technology can have business logic code mixed up with code that deals with specific details of that database technology. That is not an issue until the underlying database technology changes, forcing a code refactoring to ensure the business logic code will work with the new database technology. How can we tackle this issue? Employing hexagonal architecture may be the answer. Let us check the reason.

Alister Cockburn conceived the hexagonal architecture as a solution to develop applications where the business logic code can evolve without dependency on external technology details. Assuming that the business logic code represents the most important, often unreplaceable, asset in a software system, the hexagonal architecture application is designed to ensure the business logic code is shielded by any changes in the code responsible for dealing with any underlying technology dependency. Using hexagonal architecture allows us to develop more change-tolerable applications because we can make these applications work efficiently with different technologies. This idea of the ease of working with any technology can also be captured by the alternative name of hexagonal architecture, also known as ports and adapters architecture, where the ports express the behaviors an application supports, and the adapters express the different ways or technologies to trigger such behaviors.

You may be wondering why hexagonal architecture is named such. Its name comes from the idea that the hexagon sides represent the boundaries between the hexagonal application and the other systems in which it interacts. The hexagon form was used to differentiate from other architectural pictures, which often used rectangles to represent users and systems, and because hexagons were easier to draw than pentagons or heptagons, for example. Each hexagon side acts as either an input or output adapter. It does not mean, however, that a hexagonal application must have only six adapters, which is the number of sides a hexagon has. The hexagon shape conveys that an application can have as many adapters as possible to be compatible with different technologies. The adapters play a fundamental role by communicating between the hexagonal

application and all systems it needs to interact with to conduct its activities.

To better grasp the hexagonal architecture ideas, we explore an approach where the hexagonal system is divided into the domain, application, and framework hexagons. The domain hexagon oversees providing the domain model of the software system. The application hexagon handles data in a technology-agnostic way to enable the domain model from the domain hexagon. Finally, the framework hexagon provides the mechanisms for external systems to interact with the hexagonal application. The following figure provides a high-level representation of a hexagonal application:

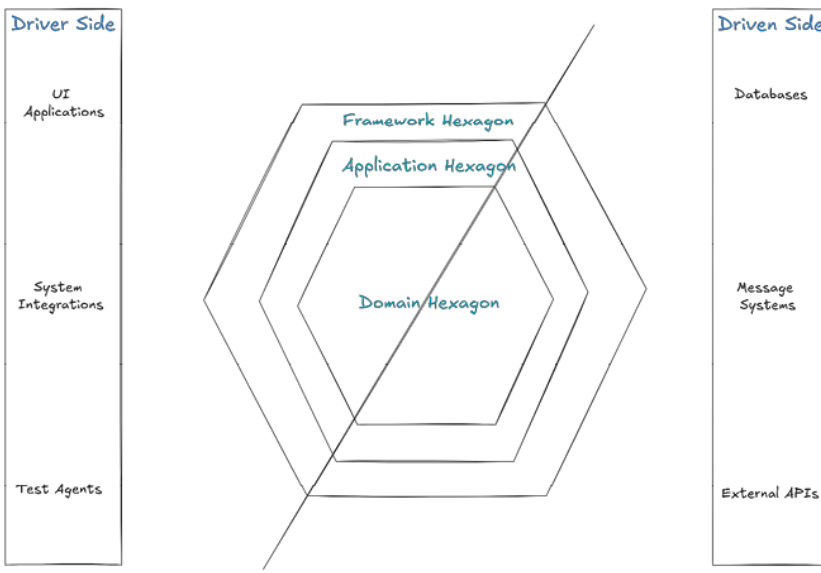

Figure 14.1: *The hexagonal application*

On the driver side, we have systems that can trigger behaviors in the hexagonal application. On the driven side, we have systems on which the hexagonal application depends. We can also use the primary and secondary terms instead of driver and driven. In the upcoming sections, we will explore the driver and driven sides further.

The domain, application, and framework hexagons each have responsibilities in the hexagonal system. Let us start exploring such responsibilities with the domain hexagon.

The domain hexagon

When starting a new software project, developers may be tempted to think first in technological terms. They may consider which software development framework best meets the project's needs, which database technology is more adequate, which caching system can provide better performance, and so on. These are all valid concerns and must be carefully considered to ensure the development of good applications. However, when these concerns are addressed at the beginning of a software project, they can influence the development process to the point where the code developed is driven more by the technologies used to solve business problems than the business problems themselves. The result is an application tailored to a specific technology stack, giving little flexibility for technology changes if necessary. Having applications optimized to work with a well-defined technology stack is not an issue if you are not expecting technology changes. However, if you expect such changes, the concerns we assessed previously regarding the technology choices of an application should be postponed to the later stages of a software project.

What should we address first if we postpone the technology choices to a later stage in a software development project? First, we address the problem domain of the application we want to develop. We start by creating code that contains the logic responsible for solving issues presented by the application's problem domain. We should strive to do that in the most technologically agnostic way. The author calls it agnostic because such a code must not depend on frameworks or external libraries that can influence how the business logic works.

The code responsible for solving business problems lives in the domain hexagon. This code can be arranged in any way; however, the domain-driven design technique is recommended because it provides the blueprint to create the domain model code, which contains all logic that solves business problems without relying on external dependencies. The following figure shows which elements are found inside the domain hexagon:

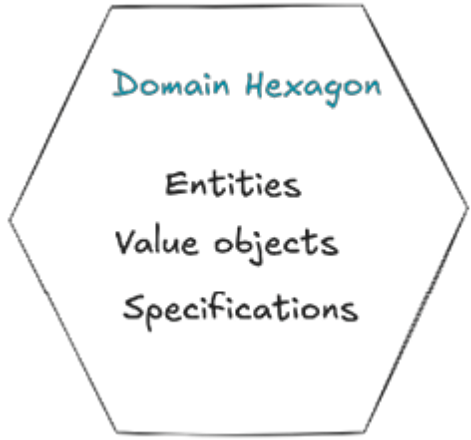

Figure 14.2: *The domain hexagon*

Relying on domain-driven design, a domain hexagon can contain entities, value objects, specifications, and other domain-driven design elements that express through code the domain model of a system. Next, we will examine some of the most essential domain-driven design elements we can use in hexagonal architecture.

Entities

In domain-driven design, we use entities to describe everything that can be uniquely identified. For example, a person, a product, a user, or an account can be uniquely identified and modeled as entities in a hexagonal application. These domain entities should not be misinterpreted with database entities. Instead, such domain entities should be designed and inspired by our knowledge of the problem domain we are dealing with. We are not concerned about which technologies we will use to handle the entities on the domain hexagon. Instead, our focus is on establishing the entity in the most straightforward way possible without depending on data or behaviors coming from outside of the application. When considering entity definition from the Java development perspective, entity classes are defined as **Plain Old Java Objects** (**POJO**), classes that rely only on the standard Java API and nothing more.

Entities carry data and behavior. For example, an account entity can use email and password as data attributes. It can also have a method called `resetPassword` as one of its behaviors.

Value objects

Contrary to entities, value objects are not uniquely identifiable. We can use, however, value objects as attributes to describe an entity. Implemented as POJOs in a Java application, value objects contribute with the domain hexagon purpose of solving business problems without depending on data and behaviors provided by third parties. Because of its immutable nature, a value object can be implemented as a record in Java. However, the author recommends using ordinary classes if the value object contains behaviors. The value objects are pure Java classes that compose classes inside the domain hexagon. For example, instead of using a String or UUID class to define an ID attribute, we can create an ID value object value containing the data and behaviors that accurately capture the identification needs of the problem domain we are working with.

Specifications

Enterprise applications are designed with a real business problem in mind. Specifications are the domain-driven design technique that lets us capture the rules for solving a system's business problems. In a hexagonal architecture application, specifications are considered the butter in the bread because they carry the most critical asset of an application, the codified business rules that solve real-world problems. Specifications are essential because understanding business rules can be challenging, requiring knowledge and experience on how a business operates. As entities and values objects, specifications are also modeled as POJOs without dependency on data or behaviors provided by third parties.

Other domain-driven design elements include aggregates and domain services; however, entities, value objects, and specifications are enough to start implementing the domain hexagon.

The domain hexagon is the foundation for the hexagonal system because it contains all the fundamental data and behaviors on which all other system parts will depend. However, the domain hexagon alone is insufficient to provide a working system. The domain hexagon becomes powerful when combined with other hexagonal architecture elements, such as ports and adapters that rely on the domain hexagon to provide fully functional features. However, it must be reinforced that the domain hexagon code must

evolve without any dependency on technological details. Such a requirement is necessary to achieve the essential outcome of shielding the code responsible for solving business problems from the code responsible for providing the technology to solve those problems.

Let us explore next how the application hexagon helps to provide data and behaviors that work based on the domain model provided by the domain hexagon through input and output ports.

The application hexagon

Back-end systems may be expected to receive and process data from users or other applications. Also, depending on the scenario, such back-end systems may need to send and retrieve data from different systems, such as databases or message brokers. On the one hand, we have actors interacting with a system by sending request payloads and triggering behaviors through the system API. On the other hand, we have the back-end system interacting with other actors responsible for providing the dependencies required to make the back-end system work. We can identify these two sides as the driver and driven sides.

The driver side comprises users and systems that interact with the hexagonal application, while the driven side corresponds to the systems on which the hexagonal application depends. Aware of this dynamic, we can affirm that the functionalities provided by a hexagonal application can be triggered on the driver side, while the dependencies to enable those functionalities come from the driven side. What lives on the driver's side has the characteristic of driving the hexagonal application by triggering its behaviors. On the other hand, what lives on the driven side is controlled and driven by the hexagonal application itself. We can implement input and output ports in the application hexagon to prepare a system to handle driver and driven operations. Also, we can employ use cases acting as abstractions for the input ports. When putting everything together, we can have input ports, output ports, and use cases as the elements that comprise the application hexagon, as shown in the following figure:

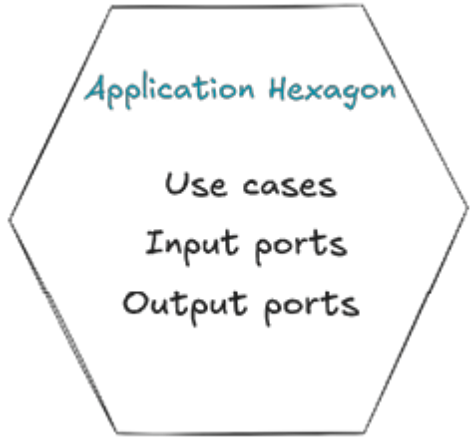

Figure 14.3: *The application hexagon*

When we start developing the application hexagon, the domain hexagon should be already implemented. The application hexagon depends on the domain hexagon to process the driver and driven operations. So, we need to determine which kind of data the hexagonal application is expected to receive from the driver side and how such data should be processed, considering the possible constraints provided by the domain hexagon. The hexagonal application may need to retrieve data from somewhere else to process the data received from the driver side. Such concerns about how the data will be received, processed, or persisted can be expressed technologically agnostic through input ports, output ports, and use cases. Let us proceed by exploring use cases.

Use cases

One of the advantages of a software system is the ability to automate things that would instead be done manually if the software did not exist. Imagine sending a message to a friend using a letter instead of email. To send a letter, we must write the message on paper, put it into an envelope, paste a postal seal, and drop it in a mailbox. These are all manual tasks we must perform to send a physical letter. Now, imagine the scenario of sending an email. The system responsible for it must provide the means to capture the message digitally written by a user. Then, it must communicate with an SMTP server to send the email message to its destination. The ability to store the email message in the system

memory and then send it using an SMTP server is an automated task provided by the software that has email delivery as one of its use cases.

The use case represents the intent of an actor using the software system. Such an actor can be a human or another system. In hexagonal applications, use cases can be defined as abstractions, through interfaces or abstract classes, that represent what the application can do. A use case abstraction may contain the operations to accomplish a given goal. In the case of an email delivery system, we can have a Java interface with two abstract methods called **`sendEmail(String message)`** and **`getSMTPServer()`** that, when used together, let the system send email messages.

If the use case is an abstraction, who implements it? The input ports are responsible for that. Let us check the input ports next.

Input ports

Input ports describe how a use case will be fulfilled. Input ports are responsible for handling data provided by clients sitting on the driver side of a hexagonal application. Such data can be processed based on the constraints provided by the domain hexagon. If necessary, the input port can use output ports to persist or retrieve data from systems on the hexagonal application's driven side.

All operations in the application hexagon are defined without specifying the technology details of systems from both the driver and driven sides of the hexagonal application. Designing the system without providing the technology details in the application hexagon gives greater flexibility because we can define system functionalities without specifying which technologies will enable those functionalities.

We learned earlier that input ports can rely on output ports to persist or retrieve data from systems on the hexagonal application's driven side. Let us explore output ports.

Output ports

Acting as abstractions, we use output ports to define what the hexagonal system needs from the outside. The idea here is to express the need to get some data without specifying how such data

will be obtained. We do that to avoid dependency on the underlying technology that provides the data. If output ports are abstractions, you may wonder who provides their implementations. That is the responsibility of the output adapter, which we will learn next while exploring the framework hexagon.

The framework hexagon

There is a moment in developing a software system in which we must decide which technologies will enable the system functionalities. When creating a new system based on the hexagonal architecture, those decisions can be postponed until the last moment when the domain model has already been implemented in the domain hexagon, and the use cases and ports are already defined in the application hexagon. What is left now is the integration of the hexagonal system with its external dependencies, which may include databases, email services, file servers, message brokers, and so on. The integration also covers providing an API that lets users and other systems send requests to the hexagonal application. The idea of having driver and driven sides, as we discussed when covering ports in the application hexagon, can also be applied when dealing with adapters in the framework hexagon.

To recap, on the driver side of the hexagonal application, we have actors such as users and other systems that can trigger system behaviors, and because of that, they are in a position to drive the hexagonal application. On the driven side, we have actors providing dependencies such as databases and other systems. It is called driven because the hexagonal application drives these actors. The input adapters support the driver side, while the driven side is handled using output adapters. The framework hexagon is composed of input and output adapters that determine which technologies are supported by the hexagonal application, as shown in the following figure:

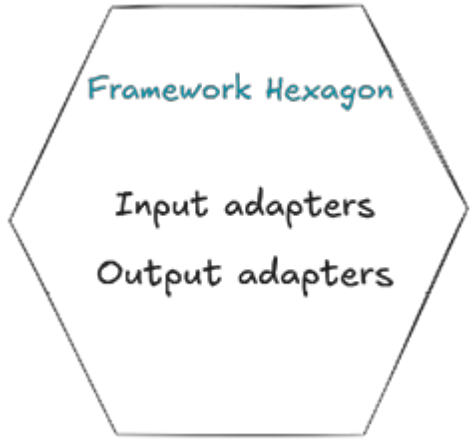

Figure 14.4: *The framework hexagon*

Next, we explore what input and output adapters are and how we can use them to make hexagonal applications compatible with any technology.

Input adapters

Once we have the hexagon application features defined as use cases and implemented as input ports, we may want to expose those features to the outside world. We accomplish that using input adapters that provide an interface enabling driver actors to interact with the hexagonal application. Input adapters resemble physical adapters in a way that both share the purpose of fitting the format of something into another thing. From the software system perspective, a format can be seen as a protocol supported by a hexagonal system. The support of a protocol is done through the definition of an API that establishes which technologies are used to communicate with the hexagonal application. We can define APIs using HTTP-based solutions like RESTful, gRPC, or SOAP. Depending on the use case, we can explore protocols like FTP for file transfer or SMTP for email delivery.

Hexagonal architecture allows us to support as many technologies as we want in the form of input adapters. A hexagonal application can be initially designed to provide an input adapter supporting RESTful requests. A new input adapter supporting gRPC calls can be quickly introduced as the application matures. Adding new input

adapters impacts only the framework hexagon. The code on the application and domain hexagons are entirely protected from changes on the framework hexagon. One or more input adapters can be connected to the same input port, which allows access to the same system functionality provided by the input port and exposes it through different technologies supported by the input adapters.

Input adapters are the hexagonal architecture elements that open the door for those interested in the functionalities offered by the hexagonal application. However, to enable those functionalities, the hexagonal application needs to get data and access systems backed by different technologies. Access to those systems is made possible through the usage of output adapters. Let us explore them next.

Output adapters

Most back-end applications depend on other systems to conduct their activities. Such dependency can be expressed by the need to access, for example, an external API responsible for processing payments, a scheduler system that executes tasks in a pre-determined time, a message broker that allows asynchronous processing, or a relation database for data persistence. Countless scenarios can be used as dependency examples for a back-end application. The critical concept here is that every dependency on something outside the hexagonal application is handled through an output adapter.

Output adapters support the driven side of a hexagonal system. They are direct implementations of the output ports defined in the application hexagon. Through the output adapters, we define the code responsible for dealing with whatever technology is necessary to provide the data or behavior to enable the hexagonal application functionalities. The system can start persisting data in MySQL databases and evolve without major refactoring to support Oracle databases. Every new technology supported is a new output adapter implementing the same output port that expresses which kind of data, based on the domain hexagon, the hexagonal system requires.

Having covered the fundamental hexagonal architecture ideas, let us see how we can apply the concepts learned in developing a hexagonal application next. We start by arranging the domain model in the domain hexagon.

Arranging the domain model

In this section, we implement a note keeper system based on the hexagonal architecture concepts we explored in the previous section. The note keeper system is a Spring Boot application that can be accessible in two ways: **command line** (**CLI**) and REST API. As the first step, we must arrange the domain model in the domain hexagon to develop such an application. Let us create the **`Note`** domain entity:

```
@Builder
@Getter
@ToString
public class Note {

    private Id id;
    private Title title;
    private String content;
    private Instant creationTime;

    private Note(Id id, Title title, String
content, Instant creationTime) {
        this.id = id;
        this.title = title;
        this.content = content;
        this.creationTime = creationTime;
    }

    public static Note of(Title title, String
content) {
        return new Note(Id.withoutId(), title,
content, Instant.now());
    }
```

```
    public static Note of(Id id, Title title,
String content, Instant
    creationTime) {
        return new Note(id, title, content,
creationTime);
    }
}
```

We use the **`@Builder`**, **`@Getter`**, and **`@ToString`** Lombok annotations to make the code more concise. Using helper libraries on domain entities can be tolerated, depending on how pure we want our domain entities to be. In cases where dependency on helper libraries is unacceptable, we can rely only on the Java standard API. For other, less strict scenarios, relying on libraries like *Lombok* can bring benefits without significantly impacting the architecture.

The static **`of(Title title, String content)`** method is used when a new Note entity needs to be created. We use the static **`of(Id id, Title title, String content, Instant creationTime)`** method to reconstruct an existing **`Note`**. You may have noticed the usage of the **`Id`** and **`Title`** attributes; these are value objects used to describe the domain model better. The following code is how the ID value object can be implemented:

```
public class Id {
    private final UUID uuid;
    private Id(UUID uuid) {
        this.uuid = uuid;
    }
    public static Id withId(String id) {
        return new Id(UUID.fromString(id));
    }
    public static Id withoutId() {
        return new Id(UUID.randomUUID());
    }
```

```
    @Override
    public String toString() {
        return uuid.toString();
    }
}
```

We use the **`Id`** class as a wrapper for the Java **`UUID`** type. **`Id`** objects can only be created by one of the available static methods **`withId`** and **`withoutId`**.

The **`Title`** is the second value object that can be implemented as follows:

```
@Getter
public class Title {

    private final String name;
    private static final int MAX_TITLE_CHARACTERS
= 120;

    private Title(String name) {
        if (name.length() > MAX_TITLE_CHARACTERS)
{
            throw new
IllegalArgumentException("Title name exceeds
            maximum character limit
"+MAX_TITLE_CHARACTERS);
        }
        this.name = name;
    }
    public static Title of(String name) {
        return new Title(name);
    }
```

```
    @Override
    public String toString() {
        return name;
    }
}
```

A **`Title`** value object can only be created through its static **`of(String name)`** method. Using the **`Title`** value object also allows us to enforce a constraint through the class' constructor, preventing the creation of titles that are too long.

The **`Note`** entity, the **`Id`**, and the **`Title`** value objects comprise the domain model provided by the domain hexagon. As we will see in upcoming sections, the domain hexagon plays a fundamental role in the hexagonal architecture because the other hexagons, namely application and framework, depend on it.

Next, let us see how input and output ports are implemented in the application hexagon.

Providing input and output ports

Inside the application hexagon, we can describe the operations that let us store and retrieve notes in a technology-agnostic way. Let us start by defining the **`NoteOutputPort`** interface:

```
public interface NoteOutputPort {

    Note persistNote(Note note);

    List<Note> getNotes();
}
```

The **`NoteOutputPort`** interface definition shows that the application hexagon depends on the domain hexagon by using the Note entity in the **`persistNote`** and **`getNotes`** methods. We do not need to know how the system will persist and get **`Note`** objects at the application hexagon level, so we express this through the **`NoteOutputPort`** interface.

The next step is to define the **`NoteUseCase`** interface:

```
public interface NoteUseCase {

    void createNote(Title title, String content);

    List<Note> getNotes();
}
```

Use cases are employed to define the behaviors supported by an application. The note keeper system allows us to create and see existing notes. These behaviors are expressed through the **`createNote`** and **`getNotes`** methods. Based on the **`NoteUseCase`** interface, we can implement **`NoteInputPort`** as follows:

```
@Service
public class NoteInputPort implements NoteUseCase
{

    private final NoteOutputPort noteOutputPort;

    public NoteInputPort (NoteOutputPort
noteOutputPort) {
        this.noteOutputPort = noteOutputPort;
    }
    @Override
    public void createNote(Title title, String
content) {
        var note = Note.of(title, content);
        persistNote(note);
    }

    private void persistNote(Note note) {
        noteOutputPort.persistNote(note);
```

```
    }

    @Override
    public List<Note> getNotes() {
        return noteOutputPort.getNotes();
    }
}
```

We use the **`@Service`** annotation to make the **`NoteInputPort`** a managed bean object controlled by the Spring Boot. Note that we are injecting **`NoteOutputPort`** as a dependency on the **`NoteInputPort`** constructor. That is when things get interesting because **`NoteOutputPort`** is defined as an interface, which means there may be different interface implementations enabling us to get the required data to fulfill the operations of the **`NoteInputPort`**. Moving ahead in the code, we implement the **`createNote`** method, which creates a **`Note`** object using the `title` and `content` attributes. Then, we persist the **`Note`** object using the **`persistNote`** method, which relies on the **`NoteOutputPort`**. Finally, we implement the **`getNotes`** method, which uses **`NoteOutputPort`** to get all stored **`Notes`**.

The **`NoteInputPort`** is based on its **`NoteUseCase`** interface, the **`NoteOutputPort`** abstraction it uses to get data from outside, and the domain model provided by the domain hexagon. How can we use the **`NoteInputPort`**? Let us discover it next when exposing input and output adapters in the framework hexagon.

Exposing input and output adapters

In the framework hexagon, we can make decisions such as how the system will be accessed and how data will be stored. We can store data in a file-based H2 database for the note keeper system. The following code is how we configure the H2 database in the **`application.yml`** file from Spring Boot:

```
spring:
  datasource:
    url: jdbc:h2:file:./note-h2-db
```

```
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    database-platform:
org.hibernate.dialect.H2Dialect
    hibernate:
      ddl-auto: update
```

The **note-h2-db** file is created when the application starts for the first time. The note keeper data will be persisted in that file. It is worth noting that the usage of the **jpa.hibernate.ddl-auto: update** property to ensure the database schema is kept updated based on the application's ORM entity definition. Let us explore such an entity definition by implementing the output adapter of the note keeper system.

Creating the output adapter

We cannot rely on the **Note** entity class defined by the domain hexagon to persist data into the database. For that purpose, we need to implement a proper Jakarta entity:

```
@Builder
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "note")
public class NoteData {

    @Id
    private String id;
    private String title;
    private String content;
```

```
    private Instant creationTime;
}
```

We use the **`@Builder`**, **`@Getter`**, **`@AllArgsConstructor`**, and **`@NoArgsConstructor`** Lombok annotations to make the code more concise. The **`NoteData`** is a Jakarta entity representing the **`Note`** domain entity defined in the domain hexagon. As we have these two different types of entities, we need a mapper mechanism that lets us convert one entity into another:

```
public class NoteMapper {


    public static Note noteDataToDomain(NoteData
noteData) {
        return Note.of(

Id.withId(noteData.getId().toString()),
                Title.of(noteData.getTitle()),
                noteData.getContent(),
                noteData.getCreationTime()
        );
    }
    public static NoteData noteDomainToData(Note
note) {
        return NoteData.builder().
                id(note.getId().toString()).
                title(note.getTitle().getName()).
                content(note.getContent()).

creationTime(note.getCreationTime())
                .build();
    }
}
```

The **NoteMapper** provides the **noteDataToDomain** and **noteDomainToData** helper methods that produce **Note** and **NoteData** objects, respectively. We will use **NodeMapper** when implementing the output adapter class later in this section. Before doing so, let us first define the repository interface to enable us to handle **NoteData** Jakarta entities:

```
@Repository

public interface NoteRepository extends
CrudRepository<NoteData, String> {

}
```

The **NoteData**, **NoteMapper**, and **NoteRepository** are the dependencies we need to implement the **NoteH2Adapter**:

```
@Component

public class NoteH2Adapter implements
NoteOutputPort {


    private final NoteRepository noteRepository;


    public NoteH2Adapter(NoteRepository
noteRepository) {

        this.noteRepository = noteRepository;

    }

    @Override

    public Note persistNote(Note note) {

        var noteData =
NoteMapper.noteDomainToData(note);

        var persistedNoteData =
noteRepository.save(noteData);

        return
NoteMapper.noteDataToDomain(persistedNoteData);

    }

    @Override
```

```
    public List<Note> getNotes() {
        var allNoteData =
noteRepository.findAll();
        var notes = new ArrayList<Note>();
        allNoteData.forEach(noteData -> {
            var note =
NoteMapper.noteDataToDomain(noteData);
            notes.add(note);
        });
        return notes;
    }
}
```

We use the **`@Component`** Spring annotation to make the **`NoteH2Adapter`** class a managed bean controlled by Spring Boot. The **`NoteH2Adapter`** implements the **`NoteOutputPort`** by implementing the **`persistNote`** and **`getNotes`** methods. The **`persistNote`** uses the **`NoteMapper`** to convert the **`Note`** domain entity to the **`NoteData`** Jakarta entity, then saves it into the database using the **`NoteRepository`**. The **`getNotes`** also relies on the **`NoteMapper`** to retrieve and convert the **`NoteData`** Jakarta entity to the domain entity format.

Such conversions on the output adapter represent the system's fundamental change-tolerable capability. **`NoteH2Adapter`** is just one way that the system can persist data. Using output ports and adapters lets us introduce new output adapters with their proper data conversions whenever necessary.

Now that we have implemented the note keeper system's output adapter, let us implement the input adapters.

Creating input adapters

We can handle notes using a CLI interface or a REST API in the note keeper system. We need to implement two input adapters to allow handling notes in different ways. One for the CLI interface and another for the REST API. Let us start implementing the **`NoteCLIAdapter`**:

```
@Component
public class NoteCLIAdapter {

    private final NoteUseCase noteUseCase;

    public NoteCLIAdapter(NoteUseCase noteUseCase)
{
        this.noteUseCase = noteUseCase;
    }
    public String createNote(Scanner
requestParams) {
        var noteParams =
stdinParams(requestParams);
        var title = noteParams.get("title");
        var content = noteParams.get("content");
        noteUseCase.createNote(Title.of(title),
content);
        return "Note created with success";
    }
    private Map<String, String>
stdinParams(Scanner requestParams) {
        Map<String, String> params = new
HashMap<>();
        System.out.println("Provide the note
title:");
        var title = requestParams.nextLine();
        params.put("title", title);
        System.out.println("Provide the note
content:");
        var content = requestParams.nextLine();
        params.put("content", content);
```

```
        return params;
    }
    public void printNotes() {

noteUseCase.getNotes().forEach(System.out::println);
    }
}
```

The **`NoteCLIAdapter`** relies on the **`NoteUseCase`** interface that the **`NoteInputPort`** implements. The input port provides the behaviors the system supports, and an adapter can trigger them. The **`createNote`** method uses the **`Scanner`** object to read data from the user's keyboard. The user-provided data is captured through the **`stdinParams`** method. Finally, we have the **`printNotes`** method, which displays all stored notes.

As an alternative to the **`NoteCLIAdapter`**, we implement the **`NoteRestAdapter`**:

```
@RestController
public class NoteRestAdapter {

    private final NoteUseCase noteUseCase;

    @Autowired
    NoteRestAdapter(NoteUseCase noteUseCase) {
        this.noteUseCase = noteUseCase;
    }
    @PostMapping("/note")
    private void addNote(@RequestBody NotePayload
notePayload) {

noteUseCase.createNote(Title.of(notePayload.title()),
        notePayload.content());
    }
```

```
    @GetMapping("/notes")
    private List<Note> all() {
        return noteUseCase.getNotes();
    }
}
```

Relying on the **`NoteUseCase`** as well, the **`NoteRestAdapter`** provides the HTTP POST **`/note`** endpoint, which lets us create new notes in the system. It also provides the HTTP GET **`/notes`** endpoint, which brings all stored notes.

The **`NoteCLIAdapter`** and **`NoteRestAdapter`** connect to the **`NoteInputPort`**, derived from the **`NoteUseCase`** interface, to provide the same behavior but through different means.

For the **`NoteCLIAdapter`** and **`NoteRestAdapter`** adapters to function effectively, it is necessary to configure Spring Boot properly. To enable the system to work in the CLI mode, we need to implement the **`NoteKeeperCLIApplication`** class:

```
@Component
@ConditionalOnNotWebApplication
public class NoteKeeperCLIApplication implements
CommandLineRunner {

    private final NoteCLIAdapter noteCLIAdapter;

    public NoteKeeperCLIApplication(NoteCLIAdapter
noteCLIAdapter) {
        this.noteCLIAdapter = noteCLIAdapter;
    }

    @Override
    public void run(String... args) {
        var operation = args[0];
        Scanner scanner = new Scanner(System.in);
```

```
        switch (operation) {
            case "createNote" –
>noteCLIAdapter.createNote(scanner);
            case "printNotes" ->
noteCLIAdapter.printNotes();
            default -> throw new
InvalidParameterException("The supported
            operations are: createNote and
getNotes");
        }
    }
}
```

We implement the **`CommandLineRunner`** interface from Spring, which enables us to execute the application in the CLI mode.

The **`run`** method utilizes the String **`varargs`** parameter to receive data passed as a parameter to the Java program, enabling it to either create a new note or print stored notes.

To enable the system to work in the web mode, we need to implement the **`NoteKeeperWebApplication`** class. The code is as follows:

```
@SpringBootApplication
public class NoteKeeperWebApplication {

    public static void main(String... args) {

SpringApplication.run(NoteKeeperWebApplication.class,
args);
    }
}
```

The only requirement to make the application run in the web mode is to use the **`@SpringBootApplication`** annotation.

To wrap up, let us see how to compile and run the note keeper

system.

Compiling and running the sample project

In this section, we compile and run the note keeper project we have been exploring in the previous sections.

You can clone the application source code from the GitHub repository at **https://github.com/bpbpublications/Java-Real-World-Projects/tree/main/Chapter%2014**.

You need the JDK 21 or above and Maven 3.8.5 or above installed on your machine.

Execute the following command to compile the application:

```
$ mvn clean package
```

Maven will create a JAR file. Once we have this file, we can test the application in different modes, as covered in the following steps.

1. Below is how we can run the application in the CLI mode:

   ```
   $ java -jar -Dspring.main.web-application-type=NONE tar

   Provide the note title:

   My goal for this week

   Provide the note content:

   I want to finish reading the Java Real World Projects b
   ```

   The input provided above creates a new note in the system.

2. We can check existing notes with the following command:

   ```
   $ java -jar -Dspring.main.web-application-type=NONE tar

   Note(id=fec50f2a-afc6-46a2-a200-bc3a13aadc05, title=My
   ```

3. By executing the following command, we can start the application in the web mode, ready to receive HTTP requests:

   ```
   $ java -jar -Dspring.main.web-application-type=NONE tar
   ```

4. After having the application running in the web mode, the following command lets us create a new note:

   ```
   $ curl -X POST localhost:8080/note -H 'Content-type:app
   ```

5. The following command is how we check all stored notes:

```
$ curl -s localhost:8080/notes | jq
[
  {
    "id": {
      "uuid": "fec50f2a-afc6-46a2-a200-bc3a13aadc05"
    },
    "title": {
      "name": "My goal for this week"
    },
    "content": "I want to finish reading the Java Real
    "creationTime": "2024-08-11T00:34:14.331301Z"
  },
  {
    "id": {
      "uuid": "f597f688-bd4c-4059-8d6b-997f81150943"
    },
    "title": {
      "name": "My goal for next week"
    },
    "content": "I want to create a Java project to prac
    "creationTime": "2024-08-11T00:35:09.966469Z"
  }
]
```

The output above shows the first note inserted using the application in the CLI mode and the note inserted when using the application in the web mode.

Conclusion

We reach the end of this book with the exploration of the hexagonal architecture. This software design technique lets us create change-tolerable applications by decoupling technology-related code from the code responsible for solving business problems. This final chapter covered the fundamentals of hexagonal architecture by exploring essential concepts like the domain hexagon and its role in providing the domain model based on domain-drive-design techniques. We learned how the hexagon application helps to express system behaviors in a technology-agnostic way, which gives excellent flexibility by not coupling the hexagonal system with third-party technologies. We also learned how the framework hexagon is fundamental in making the system behaviors compatible with different technologies.

We could apply the ideas shared in this chapter by implementing the note keeper system, a Spring Boot application structured using hexagonal architecture. This application shows how to apply concepts like use cases, input and output ports, and input and output adapters to create an application accessible through a CLI interface and a REST API.

As we conclude this book, the author wants to acknowledge the challenges you have faced and the knowledge you have gained in our exploration of Java. You have persevered by exploring the core Java API, the latest Java features, testing techniques, cloud-native development, observability, and software architecture. The author encourages you to continue learning, practicing, and overcoming the complexities that are quite often present in the life of a Java developer.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

account registration system 91-93
  Arrange-Act-Assert pattern 95
  external call, adding to 97, 98
  integrating, with MySQL 103-105
  testing 93, 94
Alertmanager 258, 271
  container, setting up 272, 273
  notification channels, defining 273, 274
alerts
  triggering, with Alertmanager 271
Amazon Web Services (AWS) 191
Amazon Web Services (AWS) Lambda 158
AOP principles
  advice 123
  aspect 122
  jointpoint 122
application hexagon 328, 329
  input ports 330
  output ports 330
  use cases 329, 330
application metrics
  capturing, with Prometheus 258, 259
  providing 240
application-scoped bean 144
Arrange-Act-Assert pattern 95
  assertions 95
aspect-oriented programming (AOP) 122

B

beans 113
  creating, with @Bean annotation 113-116
  creating, with Spring stereotype annotations 117
blocking IO
  handling, with reactive programming 51-53
bootable JAR 204
  of Jakarta EE application 206, 207

of Quarkus application 205, 206
of Spring Boot application 205
bounded contexts 279-281
business rules
defining, with specifications 291-295

C

CallableStament interface
store procedures, calling with 68, 69
checked exceptions 17, 18
Collections API 1
command line interface (CLI) application 126
container-based virtualization 194
Container Runtime Interface (CRI) 200
container technologies 192
Criteria API 84
CRUD application, with Spring Boot
API endpoints, exposing with controller 133, 134
configuring 130, 131
database entity, defining 131
dependencies, setting up 130
HTTP requests, sending 134, 135
implementation 129
repository, creating 131, 132
service, implementing 132, 133
CRUD app, with Quarkus
building 143
database entity, simplifying with Panache 150
data persistence, with Hibernate 148
dependency injection, with Quarkus DI 143

D

database connection
creating, with DataSource interface 62-64
creating, with DriverManager class 61, 62
creating, with JDBC API 60, 61
Data Definition Language (DDL) 60
data layer
account entity and repository, implementing 308
category entity and repository, implementing 306, 307
data handling 305, 306
data persisting 305, 306
Data Manipulation Language (DML) 60
data structures
handling, with collections 2
key-value data structures, creating with maps 9-12
non-duplicate collections, providing with set 6-8

ordered object collections, creating with lists 3-5
Date-Time APIs 24
Instant class 28
LocalDate 24-26
LocalDateTime 26
LocalTime 26
ZoneDateTime 27
dependency injection
with @Autowired 119-122
distributed tracing 222
implementing, with Spring Boot and OpenTelemetry 222, 223
distribution summaries 244
Docker 195
fundamentals 196
Docker-based applications
access, allowing with Service 214-216
application configuration, externalizing 210
application configuration, providing with ConfigMap 210, 211
deploying, on Kubernetes 209
deploying, with Deployment 212-214
kubectl, for installing Kubernetes objects 216, 217
Kubernetes objects, creating 210
Secret, for defining database credentials 211, 212
Docker containers
creating 197, 198
Docker image
creating 207-209
managing 196, 197
domain-driven design (DDD) 277-279
domain hexagon 326, 327
arranging 332-335
entities 327
input adapter, creating 340-342
input and output adapters 337
input and output ports, providing 335-337
output adapter, creating 337-340
specifications 328
testing 295-298
value objects 327
DriverManager class 61

E

EFK stack 233
setting up, with with Docker Composer 233-236
Elasticsearch 232
enterprise information system (EIS) tiers 167
enterprise information system tier 166

Enterprise Java Bean (EJB) 167
enterprise resource planning (ERP) 166
entity 288
  identity, expressing with 288-291
entity relationships
  defining 74
  many-to-many relationship 78-81
  many-to-one relationship 76
  one-to-many relationship 74-76
  one-to-one relationship 76-78
error handling, with exceptions 17
  checked exceptions 17, 18
  custom exceptions, creating 20, 21
  finally block 19
  try-with-resources 19, 20
  unchecked exceptions 18, 19
event storming 282, 283
event storm session participants
  actors 285
  aggregates 286
  commands 285
  domain events 284
  domain model 286
  event storm session, preparing 283, 284
  identifying 283

F

Fluentd 232
framework hexagon 330, 331
  input adapters 331, 332
  output adapters 332
functional interfaces 28
  Consumer 31
  Function 29, 30
  Predicate 29
  Stream 31, 32
  Supplier 30
functional programming
  with streams and lambdas 28

G

Google Cloud Platform (GCP) 191
GraalVM 159
Grafana dashboard
  building 268, 269
  creating, with application-generated metrics 268
  visualization for download size 271

visualization for file upload duration 270
visualization for number of requests per HTTP method 269, 270

H

hexagonal architecture 324, 325
application hexagon 328, 329
domain hexagon 326
framework hexagon 330
Hibernate 81
configuring 81
for handling database entities 81-83

I

Infrastructure-as-a-Service (IaaS) solution 193
inheritance 38
expectations 39-43
Instant class 28
integration tests 89
implementing, with Testcontainers 105-107
running, with Maven 107-109
intermediate operations 32
Inversion of Control (IoC) 113

J

Jakarta EE 164-167
enterprise application, building with 174, 175
Jakarta EE Core Profile specification 169, 170
Jakarta EE Platform specification 167, 168
Jakarta EE Web Profile specification 168, 169
multitiered applications, designing 165
Jakarta EE project 173, 174
Jakarta EE tiers
business tier 165
client tier 165
web tier 165
Jakarta Persistence
data handling, simplifying with 72, 73
entities, defining 73, 74
entity relationships, defining 74
Jakarta Server Faces (JSF) 165
Jakarta Server Pages (JSP) 165
Java 1
Java 2 Enterprise Edition (J2EE) 164
Java API 1
Java Archive (JAR) 170
Java Collections Framework 2, 3

Java Database Connectivity (JDBC) 59, 60
Java Development Kit (JDK) 159
Java Enterprise Edition (EE) 112
Java platform threads 49, 50
  blocking IO operations 50, 51
  limitations 50
Java Server Page (JSP) 167
java.util.Set interface 6
Java Virtual Machine (JVM) 27, 49, 158, 159, 208
JPQL
  exploring 83
JUnit 5
  account registration system 91-93
  setting up 90, 91
  using, for writing effective unit tests 90

K

Kibana 232
Kubernetes 198, 199
  architecture 199
  container runtime 200
  kube-apiserver 200
  kube-controller-manager 200
  kubelet 200
  kube-proxy 200
  kube-scheduler 199
Kubernetes objects
  ConfigMap 203, 204
  Deployment 201, 202
  Pod 201
  Secret 203, 204
  Service 203

L

lambda expression 29
layered architecture 303, 304
Linux Containers (LXC) 194
lists 3
LocalDate 24
LocalDateTime class 26
local development
  approaches 85
  with container databases 86
  with in-memory databases 85
  with remote databases 85
LocalTime class 26
Logging API

for improving application maintenance 21
formats 21-24
levels 21-24
log handlers 21-24

M

managed bean 143
many-to-many relationship 78
many-to-one relationship 76
Maven
tests, executing with 101
Micrometer 241
counters 242, 243
gauges 243
meters 242
registry 241
tags 242
timers 243, 244
MicroProfile 171, 172
Jakarta EE Core Profile specifications 172
specifications 173
MicroProfile project
API, building with Jakarta EE and MicroProfile 183,-186
data source, defining 179, 180
health checks, implementing 186, 187
Jakarta Persistence entity, implementing 180, 181
repository, implementing with EntityManager 181, 182
service class, implementing as Jakarta CDI managed bean 182
setting up 176-178
Mockito 96
external calls, mocking with 98-100
setting up, with JUnit 5 96
monitoring 220, 221

N

native applications
native image 159
writing, with Quarkus 158, 159
native executable
creating, with Quarkus 159, 160
NIO.2 API 13
NIO2, for file manipulation
files and directories, handling 14-16
paths, creating 12-14
using 12

O

object-oriented programming (OOP) 28
Object–Relational Mapping (ORM) 4, 83, 309
  technologies 72
observability 221, 222
one-to-many relationship 74
one-to-one relationship 76

P

Panache
  database entity handling, simplifying with 150
  with active record pattern 152, 153
  with repository pattern 151, 152
paravirtualization 193, 194
pattern matching 44
  for record 48, 49
  for switch statement 46, 47
  for type 45, 46
Plain Old Java Objects (POJO) 327
pointcut
  jointpoint 123
PrepapredStatement interface
  parameterized queries, executing with 67, 68
presentation layer 304, 305
  account endpoint, implementing 320, 321
  application behaviors, exposing 315
  category endpoint, implementing 319, 320
  transaction endpoint, implementing 316-318
Prometheus 258
  alerting rule, defining 273
  architecture 259
  configuring 261-264
  configuring, as Grafana data source 267
  downloading 261
  installing 261
  integrating, with Grafana 266, 267
  metrics consumers 260
  metrics exporters 260
  server 260
  setting up 261
PromQL
  exploring 264-266

Q

Quarkus
  benefits 138, 139

Quarkus project
  bootstrapping 139-143
  database entities, handling with EntityManager 149, 150
  database support, enabling 148, 149
Quarkus REST
  API implementation with 153-158

R

Remote File Converter 57
request-scoped beans 146-148
ResultSet object
  results, processing with 69-72

S

service layer
  account service, implementing 314, 315
  business rules, defining 309
  category service, implementing 312-314
  transaction service, implementing 309-312
simple distributed system
  building 223
  Collector, setting up 230, 231
  dependencies, configuring 223-226
  Docker Compose, setting up 230
  inventory service, implementing 226-228
  Jaeger, setting up 230
  report service, implementing 228-230
singleton beans 145, 146
software architecture 302, 303
software development 2
Spring 112
  fundamentals 112
Spring AOP
  using 123-125
Spring Boot Maven project, with Micrometer
  configuration 246, 247
  controller class, implementing 252-254
  File entity, implementing 247, 248
  File metrics, implementing 248-250
  File repository, implementing 248
  File service, implementing 250, 251
  metrics, enabling on file storage system 247
  setting up 245, 246
Spring Boot project 113
  bootstrapping 126
  creating, with Spring Initializr 126-129
Spring Boot with Micrometer

for implementing metrics 244
Spring context
for managing beans 113
Spring Core project 113
Spring stereotype annotations
@Component annotation 117
@Repository annotation 117
@Service annotation 117
using 117-119
Statement interface
simple queries, executing 64-67
Stream interface 31, 32
intermediate operation 33
stream source 32
terminal operation 34
stream source 32
Structured Query Language (SQL) syntax 60

T

terminal operation 32
Testcontainers
reliable integration tests, implementing with 102
setting up 102
thread 49
Title value objects 335

U

ubiquitous language 281, 282
unchecked exceptions 18, 19
unit tests 88, 89

V

value objects 287, 288
virtualization 192
container-based virtualization 194, 195
full virtualization 193
paravirtualization 193, 194
virtual threads 49
simple concurrent code, writing 53-57

W

Web Archive (WAR) 171

Y

Yet Another Markup Language (YAML) 200

Z

ZoneDateTime class 27